



Comments, Da
Variables, P
Statements,
Console, Exp
Operators, M
Strings, Sta
Containers,
Data, Functi

```
int main() {  
    using namespace std::views; //  
    auto const nums = array{0, 0, 1  
    auto const animals = array{"cat"  
  
    print(iota(0, 5) | chunk(2));  
    print(nums | chunk_by(equal_to{  
    print(iota(0, 5) | slide(3));  
    print(iota(0, 10) | stride(3));  
    print(repeat(8) | take(5));  
    print(zip_transform(plus{}, nums  
    print(zip(iota(0, 3), iota(1, 4  
    print(iota(0, 4) | adjacent<2>);  
    print(iota(0, 4) | pairwise);  
    print(iota(0, 4) | adjacent_tra
```



C++

The Comprehensive Guide

Torsten T. Will



Rheinwerk
Computing

Rheinwerk Computing

The Rheinwerk Computing series offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by the leading experts in their fields. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work.

Explore more of the Rheinwerk Computing library!



Johannes Ernesti, Peter Kaiser

Python 3: The Comprehensive Guide

2022, 1036 pages, paperback and e-book

www.rheinwerk-computing.com/5566



Jürgen Wolf

HTML and CSS: The Comprehensive Guide

2023, 814 pages, paperback and e-book

www.rheinwerk-computing.com/5695



Philip Ackermann

JavaScript: The Comprehensive Guide

2022, 982 pages, paperback and e-book

www.rheinwerk-computing.com/5554



Christian Ullenboom

Java: The Comprehensive Guide

2023, 1126 pages, paperback and e-book

www.rheinwerk-computing.com/5557



Philip Ackermann

Full Stack Web Development: The Comprehensive Guide

2023, 740 pages, paperback and e-book

www.rheinwerk-computing.com/5704

www.rheinwerk-computing.com

Torsten T. Will

C++

The Comprehensive Guide

Imprint

This e-book is a publication many contributed to, specifically:

Editor Megan Fuerst

Acquisitions Editor Hareem Shafi

German Edition Editor Almut Poll

Translation Torsten T. Will

Copyeditor Melinda Rankin

Cover Design Graham Geary

Photo Credit Shutterstock.com: 167285333/© titov Dmitriy; Midjourney.com

Layout Design Vera Brauner

Production E-Book Graham Geary

Typesetting E-Book SatzPro, Germany

We hope that you liked this e-book. Please share your feedback with us and read the [Service Pages](#) to find out how to contact us.

Library of Congress Cataloging-in-Publication Control Number: 2024039385

ISBN 978-1-4932-2626-9 (print)

ISBN 978-1-4932-2627-6 (e-book)

ISBN 978-1-4932-2628-3 (print and e-book)

© 2025 by Rheinwerk Publishing, Inc., Boston (MA)

1st edition 2025

3rd German edition published 2024 by Rheinwerk Verlag, Bonn, Germany

Contents

Preface	27
---------------	----

PART I Fundamentals

1 C++: The Comprehensive Guide	33
1.1 New and Modern	33
1.2 “Dan” Chapters	34
1.3 Presentation in This Book	35
1.4 Formatting Used	35
1.5 Let’s Talk Lingo	36
2 Programming in C++	37
2.1 Compiling	38
2.2 Translation Phases	39
2.3 Current Compilers	39
2.3.1 Gnu C++	40
2.3.2 Clang++ from LLVM	40
2.3.3 Microsoft Visual C++	40
2.3.4 Compiler in the Container	40
2.4 Development Environments	41
2.5 The Command Line under Ubuntu	43
2.5.1 Create a Program	44
2.5.2 Automate with Makefile	45
2.6 The Visual Studio Code IDE under Windows	46
2.7 Speed Up the Sample Program	53
3 C++ for Newcomers	55

4	The Basic Building Blocks of C++	63
4.1	A Quick Overview	66
4.1.1	Comments	66
4.1.2	The “include” Directive	66
4.1.3	The Standard Library	66
4.1.4	The “main()” Function	67
4.1.5	Types	67
4.1.6	Variables	67
4.1.7	Initialization	68
4.1.8	Output on the Console	69
4.1.9	Statements	69
4.2	A Detailed Walkthrough	70
4.2.1	Spaces, Identifiers, and Tokens	71
4.2.2	Comments	73
4.2.3	Functions and Arguments	74
4.2.4	Side Effect Operators	75
4.2.5	The “main” Function	76
4.2.6	Statements	78
4.2.7	Expressions	80
4.2.8	Allocations	81
4.2.9	Types	83
4.2.10	Variables: Declaration, Definition, and Initialization	89
4.2.11	Initialize with “auto”	90
4.2.12	Details on the Include Directive	92
4.2.13	Modules	93
4.2.14	Input and Output	94
4.2.15	The “std” Namespace	95
4.3	Operators	97
4.3.1	Operators and Operands	97
4.3.2	Overview of Operators	98
4.3.3	Arithmetic Operators	99
4.3.4	Bit-by-Bit Arithmetic	100
4.3.5	Composite Assignment	103
4.3.6	Post- and Preincrement and Post- and Predecrement	104
4.3.7	Relational Operators	104
4.3.8	Logical Operators	105
4.3.9	Pointer and Dereference Operators	106
4.3.10	Special Operators	107
4.3.11	Function-Like Operators	109
4.3.12	Operator Sequence	110

4.4	Built-In Data Types	111
4.4.1	Overview	112
4.4.2	Initialize Built-In Data Types	114
4.4.3	Integers	114
4.4.4	Floating-Point Numbers	127
4.4.5	Truth Values	141
4.4.6	Character Types	143
4.4.7	Complex Numbers	146
4.5	Undefined Behavior	149
 5 Good Code, 1st Dan: Writing Readable Code		151
5.1	Comments	151
5.2	Documentation	152
5.3	Indentations and Line Length	153
5.4	Lines per Function and File	154
5.5	Brackets and Spaces	154
5.6	Names	156
 6 Higher Data Types		159
6.1	The String Type “string”	160
6.1.1	Initialization	161
6.1.2	Functions and Methods	162
6.1.3	Other String Types	163
6.1.4	For Viewing Only: “string_view”	164
6.2	Streams	166
6.2.1	Input and Output Operators	166
6.2.2	“getline”	168
6.2.3	Files for Input and Output	168
6.2.4	Manipulators	170
6.2.5	The “endl” Manipulator	172
6.3	Container and Pointer	172
6.3.1	Container	172
6.3.2	Parameterized Types	173

6.4	The Simple Sequence Containers	174
6.4.1	“array”	174
6.4.2	“vector”	176
6.5	Algorithms	179
6.6	Pointers and C-Arrays	180
6.6.1	Pointer Types	180
6.6.2	C-Arrays	180

7 Functions **181**

7.1	Declaration and Definition of a Function	182
7.2	Function Type	183
7.3	Using Functions	184
7.4	Defining a Function	185
7.5	More about Parameters	186
7.5.1	Call-by-Value	186
7.5.2	Call-by-Reference	187
7.5.3	Constant References	188
7.5.4	Call as Value, Reference, or Constant Reference?	189
7.6	Functional Body	190
7.7	Converting Parameters	192
7.8	Overloading Functions	194
7.9	Default Parameter	196
7.10	Arbitrary Number of Arguments	198
7.11	Alternative Notation for Function Declaration	198
7.12	Specialties	199
7.12.1	“noexcept”	199
7.12.2	Inline Functions	200
7.12.3	“constexpr”	200
7.12.4	Deleted Functions	201
7.12.5	Specialties in Class Methods	201

8 Statements in Detail 203

8.1	The Statement Block	206
8.1.1	Standalone Blocks and Variable Scope	207
8.2	The Empty Statement	208
8.3	Declaration Statement	209
8.3.1	Structured Binding	210
8.4	The Expression Statement	211
8.5	The “if” Statement	212
8.5.1	“if” with Initializer	215
8.5.2	Compile-Time “if”	215
8.6	The “while” Loop	216
8.7	The “do-while” Loop	218
8.8	The “for” Loop	219
8.9	The Range-Based “for” Loop	221
8.10	The “switch” Statement	222
8.11	The “break” Statement	227
8.12	The “continue” Statement	228
8.13	The “return” Statement	228
8.14	The “goto” Statement	229
8.15	The “try-catch” Block and “throw”	231
8.16	Summary	232

9 Expressions in Detail 233

9.1	Calculations and Side Effects	234
9.2	Types of Expressions	235
9.3	Literals	236
9.4	Identifiers	237
9.5	Parentheses	237
9.6	Function Call and Index Access	238
9.7	Assignment	238
9.8	Type Casting	240

10	Error Handling	243
10.1	Error Handling with Error Codes	245
10.2	What Is an Exception?	248
10.2.1	Throwing and Handling Exceptions	249
10.2.2	Unwinding the Call Stack	250
10.3	Minor Error Handling	251
10.4	Throwing the Exception Again: “rethrow”	251
10.5	The Order in “catch”	252
10.5.1	No “finally”	253
10.5.2	Standard Library Exceptions	253
10.6	Types for Exceptions	254
10.7	When an Exception Falls Out of “main”	255
11	Good Code, 2nd Dan: Modularization	257
11.1	Program, Library, Object File	257
11.2	Modules	258
11.3	Separating Functionalities	259
11.4	A Modular Example Project	260
11.4.1	Namespaces	263
11.4.2	Implementation	264
11.4.3	Using the Library	270

PART II Object-Oriented Programming and More

12	From Structure to Class	275
12.1	Initialization	278
12.2	Returning Custom Types	279
12.3	Methods Instead of Functions	280
12.4	The Better “print”	283
12.5	An Output Like Any Other	285

12.6	Defining Methods Inline	286
12.7	Separate Implementation and Definition	287
12.8	Initialization via Constructor	288
12.8.1	Member Default Values in Declaration	291
12.8.2	Constructor Delegation	292
12.8.3	Default Values for Constructor Parameters	293
12.8.4	Do Not Call the “init” Method in the Constructor	294
12.8.5	Exceptions in the Constructor	295
12.9	Struct or Class?	295
12.9.1	Encapsulation	297
12.9.2	“public” and “private”, Struct and Class	297
12.9.3	Data with “struct”, Behavior with “class”	298
12.9.4	Initialization of Types with Private Data	298
12.10	Interim Recap	299
12.11	Using Custom Data Types	300
12.11.1	Using Classes as Values	302
12.11.2	Using Constructors	305
12.11.3	Type Conversions	306
12.11.4	Encapsulate and Decapsulate	308
12.11.5	Give Types a Local Name	312
12.12	Type Inference with “auto”	315
12.13	Custom Classes in Standard Containers	319
12.13.1	Three-Way Comparison: The Spaceship Operator	321
13	Namespaces and Qualifiers	323
13.1	The “std” Namespace	324
13.2	Anonymous Namespace	327
13.3	“static” Makes Local	329
13.4	“static” Likes to Share	330
13.5	Remote Initialization or “static inline” Data Fields	332
13.6	Guaranteed to Be Initialized at Compile Time with “constinit”	333
13.7	“static” Makes Permanent	333
13.8	“inline namespace”	335
13.9	Interim Recap	336

13.10 “const”	337
13.10.1 Const Parameters	338
13.10.2 Const Methods	339
13.10.3 “const” Variables	341
13.10.4 Const Returns	342
13.10.5 “const” Together with “static”	346
13.10.6 Even More Constant with “constexpr”	347
13.10.7 “if constexpr” for Compile-Time Decisions	350
13.10.8 C++20: “consteval”	352
13.10.9 “if consteval”	354
13.10.10 Un-“const” with “mutable”	355
13.10.11 Const-Correctness	355
13.10.12 Summary	357
13.11 Volatile with “volatile”	357

14 Good Code, 3rd Dan: Testing 361

14.1 Types of Tests	361
14.1.1 Refactoring	362
14.1.2 Unit Tests	363
14.1.3 Social or Solitary	364
14.1.4 Doppelgangers	366
14.1.5 Suites	367
14.2 Frameworks	368
14.2.1 Arrange, Act, Assert	370
14.2.2 Frameworks to Choose From	371
14.3 Boost.Test	372
14.4 Helper Macros for Assertions	376
14.5 An Example Project with Unit Tests	379
14.5.1 Private and Public Testing	380
14.5.2 An Automatic Test Module	381
14.5.3 Compile Test	384
14.5.4 Assemble the Test Suite Yourself	384
14.5.5 Testing Private Members	388
14.5.6 Parameterized Tests	389

15 Inheritance 391

15.1 Relationships	392
15.1.1 Has-a Composition	392
15.1.2 Has-a Aggregation	392
15.1.3 Is-a Inheritance	393
15.1.4 Instance-of versus Is-a Relationship	394
15.2 Inheritance in C++	394
15.3 Has-a versus Is-a	395
15.4 Finding Commonalities	396
15.5 Derived Types Extend	398
15.6 Overriding Methods	399
15.7 How Methods Work	400
15.8 Virtual Methods	402
15.9 Constructors in Class Hierarchies	404
15.10 Type Conversion in Class Hierarchies	405
15.10.1 Converting Up the Inheritance Hierarchy	405
15.10.2 Downcasting the Inheritance Hierarchy	406
15.10.3 References Also Retain Type Information	406
15.11 When to Use Virtual?	407
15.12 Other Designs for Extensibility	409

16 The Lifecycle of Classes 411

16.1 Creation and Destruction	412
16.2 Temporary: Short-Lived Values	414
16.3 The Destructor to the Constructor	416
16.3.1 No Destructor Needed	418
16.3.2 Resources in the Destructor	418
16.4 Yoda Condition	420
16.5 Construction, Destruction, and Exceptions	421
16.6 Copy	423
16.7 Assignment Operator	426
16.8 Removing Methods	429

16.9 Move Operations	430
16.9.1 What the Compiler Generates	434
16.10 Operators	435
16.11 Custom Operators in a Data Type	438
16.12 Special Class Forms	446
16.12.1 Abstract Classes and Methods	446
16.12.2 Enumeration Classes	448

17 Good Code, 4th Dan: Security, Quality, and Sustainability

17.1 The Rule of Zero	451
17.1.1 The Big Five	451
17.1.2 Helper Construct by Prohibition	452
17.1.3 The Rule of Zero and Its Application	453
17.1.4 Exceptions to the Rule of Zero	454
17.1.5 Rule of Zero, Rule of Three, Rule of Five, Rule of Four and a Half	456
17.2 Resource Acquisition Is Initialization	457
17.2.1 An Example with C	457
17.2.2 Owning Raw Pointers	459
17.2.3 From C to C++	460
17.2.4 It Doesn't Always Have to Be an Exception	462
17.2.5 Multiple Constructors	463
17.2.6 Multiphase Initialization	464
17.2.7 Define Where It Is Needed	464
17.2.8 “new” without Exceptions	464

18 Specials for Classes

18.1 Allowed to See Everything: “friend” Classes	467
18.1.1 “friend class” Example	469
18.2 Nonpublic Inheritance	471
18.2.1 Impact on the Outside World	473
18.2.2 Nonpublic Inheritance in Practice	475
18.3 Signature Classes as Interfaces	477
18.4 Multiple Inheritance	481

18.4.1	Multiple Inheritance in Practice	483
18.4.2	Caution with Pointer Type Conversions	487
18.4.3	The Observer Pattern as a Practical Example	489
18.5	Diamond-Shaped Multiple Inheritance: “virtual” for Class Hierarchies	490
18.6	Literal Data Types: “constexpr” for Constructors	495

19 Good Code, 5th Dan: Classical Object-Oriented Design 497

19.1	Objects in C++	499
19.2	Object-Oriented Design	500
19.2.1	SOLID	500
19.2.2	Don't Be STUPID	516

PART III Advanced Topics

20 Pointers 521

20.1	Addresses	522
20.2	Pointer	523
20.3	Dangers of Aliasing	525
20.4	Heap Memory and Stack Memory	526
20.4.1	The Stack	526
20.4.2	The Heap	528
20.5	Smart Pointers	530
20.5.1	“unique_ptr”	532
20.5.2	“shared_ptr”	536
20.6	Raw Pointers	539
20.7	C-Arrays	543
20.7.1	Calculating with Pointers	544
20.7.2	Decay of C-Arrays	545
20.7.3	Dynamic C-Arrays	547
20.7.4	String Literals	548
20.8	Iterators	550

20.9	Pointers as Iterators	551
20.10	Pointers in Containers	552
20.11	The Exception: When Cleanup Is Not Necessary	552
 21 Macros		555
21.1	The Preprocessor	556
21.2	Beware of Missing Parenthesis	560
21.3	Feature Macros	561
21.4	Information about the Source Code	561
21.5	Warning about Multiple Executions	562
21.6	Type Variability of Macros	563
21.7	Summary	566
 22 Interface to C		567
22.1	Working with Libraries	568
22.2	C Header	569
22.3	C Resources	572
22.4	“void” Pointers	572
22.5	Reading Data	573
22.6	The Main Program	575
22.7	Summary	575
 23 Templates		577
23.1	Function Templates	578
23.1.1	Overloading	579
23.1.2	A Type as Parameter	580
23.1.3	Function Body of a Function Template	580
23.1.4	Values as Template Parameters	583
23.1.5	Many Functions	584

23.1.6	Parameters with Extras	585
23.1.7	Method Templates are Just Function Templates	587
23.2	Function Templates in the Standard Library	588
23.2.1	Ranges instead of Containers as Template Parameters	589
23.2.2	Example: Information about Numbers	592
23.3	A Class as a Function	593
23.3.1	Values for a “Function” Parameter	594
23.3.2	C Function Pointer	595
23.3.3	The Somewhat Different Function	597
23.3.4	Practical Functors	600
23.3.5	Algorithms with Functors	602
23.3.6	Anonymous Functions: a.k.a. Lambda Expressions	602
23.3.7	Template Functions without “template”, but with “auto”	608
23.4	C++ Concepts	609
23.4.1	How to Read Concepts	609
23.4.2	How to Use Concepts	612
23.4.3	How to Write Concepts	614
23.4.4	Semantic Constraints	615
23.4.5	Interim Recap	616
23.5	Template Classes	616
23.5.1	Implementing Class Templates	617
23.5.2	Implementing Methods of Class Templates	618
23.5.3	Creating Objects from Class Templates	620
23.5.4	Class Templates with Multiple Formal Data Types	623
23.5.5	Class Templates with Nontype Parameters	625
23.5.6	Class Templates with Defaults	626
23.5.7	Specializing Class Templates	628
23.6	Templates with Variable Argument Count	630
23.6.1	“sizeof ...” Operator	633
23.6.2	Convert Parameter Pack to Tuple	633
23.7	Custom Literals	634
23.7.1	What Are Literals?	635
23.7.2	Naming Rules	635
23.7.3	Literal Processing Phases	636
23.7.4	Overloading Variants	636
23.7.5	User-Defined Literal Using Template	638
23.7.6	Raw or Cooked	641
23.7.7	Automatically Merged	642
23.7.8	Unicode Literals	642

PART IV The Standard Library

24 Containers	647
24.1 Basics	648
24.1.1 Recurring	648
24.1.2 Abstract	649
24.1.3 Operations	650
24.1.4 Complexity	651
24.1.5 Containers and Their Iterators	653
24.1.6 Ranges Simplify Iterators	656
24.1.7 Ranges, Views, Concepts, Adapters, Generators, and Algorithms	659
24.1.8 Algorithms	660
24.2 Iterator Basics	660
24.2.1 Iterators from Containers	662
24.2.2 More Functionality with Iterators	663
24.3 Allocators: Memory Issues	665
24.4 Container Commonalities	668
24.5 An Overview of the Standard Container Classes	669
24.5.1 Type Aliases of Containers	670
24.6 The Sequential Container Classes	673
24.6.1 Commonalities and Differences	675
24.6.2 Methods of Sequence Containers	677
24.6.3 “vector”	679
24.6.4 “array”	698
24.6.5 “deque”	704
24.6.6 “list”	708
24.6.7 “forward_list”	711
24.7 Associative and Ordered	716
24.7.1 Commonalities and Differences	717
24.7.2 Methods of Ordered Associative Containers	718
24.7.3 “set”	719
24.7.4 “map”	733
24.7.5 “multiset”	740
24.7.6 “multimap”	745
24.8 Only Associative and Not Guaranteed	749
24.8.1 Hash Tables	749
24.8.2 Commonalities and Differences	754
24.8.3 Methods of Unordered Associative Containers	756

24.8.4	“unordered_set”	757
24.8.5	“unordered_map”	766
24.8.6	“unordered_multiset”	770
24.8.7	“unordered_multimap”	776
24.9	Container Adapters	779
24.10	Special Cases: “string”, “basic_string”, and “vector<char>”	781
24.11	Special Cases: “vector<bool>”, “array<bool,n>”, and “bitset<n>”	782
24.11.1	Dynamic and Compact: “vector<bool>”	783
24.11.2	Static: “array<bool,n>” and “bitset<n>”	783
24.12	Special Case: Value Array with “valarray<>”	786
24.12.1	Element Properties	787
24.12.2	Initialize	789
24.12.3	Assignment	789
24.12.4	Insert and Delete	790
24.12.5	Accessing	790
24.12.6	Specialty: Manipulate All Data	790
24.12.7	Specialty: Slicing and Masking	791

25 Container Support 795

25.1	Algorithms	796
25.2	Iterators and Ranges	798
25.3	Iterator Adapter	800
25.4	Algorithms of the Standard Library	800
25.5	Parallel Execution	802
25.6	Lists of Algorithm Functions and Range Adapters	805
25.6.1	Range Adapters and Views	806
25.6.2	Ranges as Parameters (and More)	813
25.6.3	List of Nonmodifying Algorithms	815
25.6.4	Inherently Modifying Algorithms	820
25.6.5	Algorithms for Partitions	825
25.6.6	Algorithms for Sorting and Fast Searching in Sorted Ranges	826
25.6.7	Set Algorithms Represented by a Sorted Range	827
25.6.8	Heap Algorithms	829
25.6.9	Minimum and Maximum	830
25.6.10	Various Algorithms	830

25.7	Element-Linking Algorithms from “<numeric>” and “<ranges>”	831
25.8	Copy instead of Assignment: Values in Uninitialized Memory Areas	838
25.9	Custom Algorithms	840
25.10	Writing Custom Views and Range Adapters	842
26	Good Code, 6th Dan: The Right Container for Each Task	845
26.1	All Containers Arranged by Aspects	845
26.1.1	When Is a “vector” Not the Best Choice?	845
26.1.2	Always Sorted: “set”, “map”, “multiset”, and “multimap”	846
26.1.3	In Memory Contiguously: “vector”, “array”	846
26.1.4	Cheap Insertion: “list”	847
26.1.5	Low Memory Overhead: “vector”, “array”	848
26.1.6	Size Dynamic: All Except “array”	849
26.2	Recipes for Containers	850
26.2.1	Two Phases? “vector” as a Good “set” Replacement	850
26.2.2	Output the Contents of a Container to a Stream	852
26.2.3	So “array” Is Not That Static	852
26.3	Implementing Algorithms That Are Specialized Depending on the Container	856
27	Streams, Files, and Formatting	857
27.1	Input and Output Concept with Streams	857
27.2	Global, Predefined Standard Streams	858
27.2.1	Stream Operators << and >>	859
27.3	Methods for Stream Input and Output	860
27.3.1	Methods for Unformatted Output	860
27.3.2	Methods for Unformatted Input	862
27.4	Error Handling and Stream States	864
27.4.1	Methods for Handling Stream Errors	865
27.5	Manipulating and Formatting Streams	867
27.5.1	Manipulators	868
27.5.2	Creating Custom Manipulators without Arguments	873

27.5.3	Creating Custom Manipulators with Arguments	875
27.5.4	Directly Change Format Flags	876
27.6	Streams for File Input and Output	879
27.6.1	The “ifstream”, “ofstream”, and “fstream” Streams	879
27.6.2	Connecting to a File	879
27.6.3	Reading and Writing	884
27.6.4	Random Access	890
27.6.5	Synchronized Streams for Threads	891
27.7	Streams for Strings	892
27.7.1	Difference from “to_string”	896
27.7.2	“to_chars” and “format” Are More Flexible than “to_string”	897
27.7.3	Reading from a String	897
27.8	Stream Buffers	898
27.8.1	Access to the Stream Buffer of “iostream” Objects	899
27.8.2	“filebuf”	900
27.8.3	“stringbuf”	900
27.9	“filesystem”	900
27.10	Formatting	902
27.10.1	Simple Formatting	903
27.10.2	Formatting Custom Types	905

28 Standard Library: Extras 909

28.1	“pair” and “tuple”	909
28.1.1	Returning Multiple Values	910
28.2	Regular Expressions	917
28.2.1	Matching and Searching	918
28.2.2	The Result and Parts of It	918
28.2.3	Found Replacement	919
28.2.4	Rich in Variants	919
28.2.5	Iterators	920
28.2.6	Matches	920
28.2.7	Options	921
28.2.8	Speed	921
28.2.9	Standard Syntax, Slightly Shortened	922
28.2.10	Notes on Regular Expressions in C++	923

28.3 Randomness	926
28.3.1 Rolling a Die	927
28.3.2 True Randomness	929
28.3.3 Other Generators	929
28.3.4 Distributions	931
28.4 Mathematical	935
28.4.1 Fraction and Time: “<ratio>” and “<chrono>”	935
28.4.2 Predefined Suffixes for User-Defined Literals	957
28.5 System Error Handling with “system_error”	960
28.5.1 Overview	960
28.5.2 Principles	961
28.5.3 “error_code” and “error_condition”	962
28.5.4 Error Categories	966
28.5.5 Custom Error Codes	966
28.5.6 “system_error” Exception	967
28.6 Runtime Type Information: “<typeinfo>” and “<typeid>”	968
28.7 Helper Classes around Functors: “<functional>”	972
28.7.1 Function Objects	973
28.7.2 Function Generators	977
28.8 “optional” for a Single Value or No Value	980
28.9 “variant” for One of Several Types	980
28.10 “any” Holds Any Type	982
28.11 Special Mathematical Functions	983
28.12 Fast Conversion with “<charconv>”	984

29 Threads: Programming with Concurrency 987

29.1 C++ Threading Basics	988
29.1.1 Starting Pure Threads	989
29.1.2 Terminating a Thread Prematurely	990
29.1.3 Waiting for a Thread	991
29.1.4 Consider Exceptions in the Starting Thread	992
29.1.5 Passing Parameters to a Thread Function	995
29.1.6 Moving a Thread	1000
29.1.7 How Many Threads to Start?	1002
29.1.8 Which Thread Am I?	1004

29.2 Shared Data	1005
29.2.1 Data Races	1005
29.2.2 Latch	1008
29.2.3 Barriers	1008
29.2.4 Mutexes	1010
29.2.5 Interface Design for Multithreading	1012
29.2.6 Locks Can Lead to Deadlock	1017
29.2.7 More Flexible Locking with “unique_lock”	1019
29.3 Other Synchronization Options	1021
29.3.1 Call Only Once with “once_flag” and “call_once”	1021
29.3.2 Locking with “recursive_mutex”	1023
29.4 In Its Own Storage with “thread_local”	1024
29.5 Waiting for Events with “condition_variable”	1025
29.5.1 “notify_all”	1028
29.5.2 Synchronize Output	1029
29.6 Waiting Once with “future”	1030
29.6.1 Launch Policies	1031
29.6.2 Wait Until a Certain Time	1032
29.6.3 Exception Handling with “future”	1035
29.6.4 “promise”	1037
29.7 Atomics	1040
29.7.1 Overview of the Operations	1042
29.7.2 Memory Order	1043
29.7.3 Example	1045
29.8 Coroutines	1046
29.8.1 Coroutines in the Compiler	1046
29.8.2 Generator	1047
29.8.3 Coroutines with “promise_type”	1048
29.9 Summary	1051
29.9.1 “<thread>” Header	1051
29.9.2 “<latch>” and “<barrier>” Headers	1051
29.9.3 “<mutex>” and “<shared mutex>” Headers	1052
29.9.4 “<condition variable>” Header	1052
29.9.5 “<future>” Header	1053
29.9.6 “<atomic>” Header	1053
29.9.7 “<coroutine>” Header	1053

30	Good Code, 7th Dan: Guidelines	1055
30.1	Guideline Support Library	1056
30.2	C++ Core Guidelines	1056
30.2.1	Motivation	1057
30.2.2	Type Safety	1058
30.2.3	Use RAll	1059
30.2.4	Class Hierarchies	1062
30.2.5	Generic Programming	1064
30.2.6	Do Not Be Confused by Anachronisms	1067
	Appendices	1069
A	Cheat Sheet	1069
B	The Author	1073
	Index	1075
	Service Pages	I
	Legal Notes	II

Preface

Thank you for purchasing this book! There are always modern features in C++ that enable new ways of working and new idioms. If you ask the internet for examples in C++, most sources lean towards the C-like style, which has little to do with the possibilities of *modern* C++: resource acquisition is initialization (RAII) up front, new concepts and modules and coroutines and ranges. As always, I intend to show you the strengths of C++ in particular—especially where I believe that C++ is ahead of other languages.

In this edition, I am assuming you will use a compiler that is fully capable of C++17. Because all popular compilers now also support C++20, you will have no problems with those features either. In the book, I mostly point out the use of C++20, but at the publisher's request, I have refrained from specific typographical emphasis on the same. I also describe C++23 features, which I always point out and emphasize, as compiler support for these features is far from complete.

I want to briefly draw your attention to C++17 features that I no longer emphasize, but that I use in almost all listings and that could confuse you if your compiler does not handle them. You still need to write the *class template argument deduction* for constructors in `vector data{1,2,3}` as `vector<int> data{1,2,3}`. You will miss `string_view` and instead need to write `const string&`. There are a few more little things that I won't list here. In C++20, *abbreviated function templates* have been added, which are extremely useful, but sometimes difficult to point out in listings. If you ever see `auto` in a function parameter, then it is actually a function template.

However, there are also gaps: Modules unfortunately are not yet fully developed, so I cannot give you many tips on them. And while `format` is widely supported, this is not yet the case for `print`.

Of course, there is nothing to be said against writing your program “traditionally.” In short, to me, that means that your program looks more like C than it could look. There's nothing wrong with that, of course. Some of the best programs are written in C. Nevertheless, if you start a project *today* and decide to use a programming language translated into machine code, you'd be better off using C++. Because something is happening in the language—or rather, something has happened. With C++23, you have a language that supports you in writing *good* programs in an up-to-date way. This means that your programs are fast, error resistant, and maintainable. You can program productively.

For this book, I spent a long time thinking about the best way to teach C++. Bjarne Stroustrup gave a keynote speech at CppCon 2017 that had this very topic at its core. He said things there that I think are earth-shattering—at least as far as the C++ world is concerned. He said: “We (teachers) did a poor job of teaching people C++ until C++98.” And he thought about why that was the case. He includes himself in this and summarizes

that most C++ books are long, monotonous, and slow. They teach “bottom-up 1990 C++” and use C++11 syntax. And that is wrong. Now, I started writing this book long before Bjarne Stroustrup gave this keynote—and that is precisely why I feel vindicated in the way this book now looks at the end of the work process. I see it the same way, and have tried to present C++ in a different way, rather than from the ground up.

For example, you won’t find pointers explained in this book until well into the book, which may seem like a pretty bold choice. Pointers are important, and C++ is all about addresses—but since C++11, they’re not everything. It is much more important to understand the concept behind pointers, manifested in iterators. Because if you understand the mechanism, you can combine the detail with other things and create something new. I always want to see the *why* in the foreground.

Stroustrup says in his keynote that the new C++ emphasizes resource security, among other things. He asked, which book emphasized RAI? There are few. You will come across the term *RAII* several times in this book. He criticizes that many books do not even mention type safety, abstraction, class design, and generic programming. This book does.

However, accuracy and care are also important to me, which is why there is also a more technical section here that deals with the syntax and semantics of the small and large C++ constructs. You can’t skip this in a manual. In a single project, it may be enough to know a single rule about which default operations should be defined for a class. But in an architecture and to understand it, you need to know which default operations exist and how they interact with each other. My approach is therefore to teach you things in three bite-size chunks. The first overview is done in a few pages and gives you the first feel for a C++ program. This is followed by the larger loop, in which I briefly cover almost every language element so that you understand the interactions: you’ll learn about expressions, types, statements, variables, and the standard library. Only in the third round do I go into all these elements in detail in individual chapters. There you will find things explained with background and interaction with the world: bits, bytes, big-endian, floating-point formats, exceptions, classes, and so on.

Especially close to my heart are the chapters on *vector*, *map*, and the like—that is, the topic of *containers*. The containers in the standard library are underestimated and consistently underused. Why is that? Back to Bjarne: Because we haven’t communicated it well enough. I’m trying to take a different approach here. Instead of just listing what containers there are and what interfaces and features they have, I want to draw your attention more to the concepts, especially the similarities and differences between the containers. I don’t want you to memorize all of *vector*’s methods right from the start, but rather what the containers can do, what they can’t do, and when you should choose which one. I have therefore written a separate chapter on the latter. If you have a problem, you can only find the right container with a reference *if* you have read and internalized all the container descriptions. I will therefore describe typical problems and present criteria that you can use to select the right container.

And that still doesn't satisfy me. Knowing C++ doesn't automatically mean you can program. But anyone who uses the new C++ correctly has understood what it's all about. And because "what it's all about" is not only important in C++, but also in other programming languages, it is important to me that you think outside the box—as is also required in the board game of Go. In Go, beginners have kyu ranks, and more advanced players earn their dan ranks. When you follow the advice in the interspersed "dan" chapters that deal with things that occur everywhere in software development, your general programming skills will also improve. No matter whether you program in Java, PHP, or SQL, you need to test and to modularize your code, and in most cases, it doesn't hurt to know object-oriented programming (OOP). I'll cover these points and apply them with C++; take them with you on your programming and architecture journey.

During my research on the innovations in C++20 and C++23, I also used the latest techniques. This means that I, as the author, have also consulted an artificial intelligence (AI) assistant from time to time—as you should also do to stay up to date. So it may not be a given that you have decided to pick up a book. But what I've realized, especially when working with the pretty impressive AI tools, is that they have their limitations. Especially when it comes to the innovations in C++20 and C++23, AI tools have rarely been able to help me. At the moment, these tools are simply parroting what the internet has been offering since its inception.

Nevertheless, they support us in our daily work. And yes, they amaze me—despite their naivety. I am sure you are aware that you must always remain skeptical of the answers they give you. Always! And that won't get any easier in the future. Andrei Alexandrescu, who currently works at Nvidia, makes predictions in his very enlightening presentation in the closing keynote speech at Code Europe 2023 that show me where the journey is heading: AI tools will increasingly support us, AI will become more commonplace, and we will notice it less and less—and therefore question it less. But that is, of course, dangerous. Therefore, stay vigilant. And enjoy this book!

Torsten T. Will

August 2024, Bielefeld

PART I

Fundamentals

You'll need a compiler or an IDE and must understand how to use them and how they work with your computer.

Then you'll find out how the C++ language is structured and how you can put the various building blocks together into an executable program.

Chapter 1

C++: The Comprehensive Guide

With this book on your desk (because it's probably too heavy for your hands), I hope to provide a work that bridges the gap between textbook and reference. I want to give you a comprehensive guide to programming with C++ to the point that you know where to look next. This is perhaps a somewhat strange approach, but I am deliberately making two compromises here: First, I hope that you already know a little about programming in general and have possibly already gained some experience with the computer's "way of thinking." You don't necessarily need to know C++ itself, which is where this book comes in. Second, there are many details, potential uses, and interactions with other language elements for each language element—and there are many, *many* of them—so I will describe each language element, but only to a certain depth. However, I embed the descriptions in a context that helps you to develop an understanding. The pure text of the C++ language standard, including the associated standard library, comprises over 2,000 pages—tightly printed and formally written down (the document "N4971" without index). A work such as this book cannot help but present it to you in an understandable but comprehensive way at around 1,000 pages, to be supplemented by a comprehensive reference elsewhere. I recommend reference sites (<http://cppreference.com>), forums (<http://stackoverflow.com>), and searching (<http://google.com>, <http://bing.com>) on the internet. AI tools might also be useful, but be wary of them.

I have structured this book in such a way that you will first get to know the tool you are working with better. So you will get answers to the questions of what a compiler or a development environment is and how to set up each. Then you will get a quick bird's-eye overview to make it easier for you to read the later chapters.

Then it gets more detailed. First, you will get to know the language core: How is a C++ program structured, and what elements does your code contain? This is mainly about syntax and semantics; you will also see the built-in data types and learn how the computer calculates with them.

This is followed by a comprehensive description of the standard library with all its tools, but also the concepts that are important for its effective use.

1.1 New and Modern

Through all of this, you will get to know the *new, modern* C++. Why new? Because a *new, modern* era has begun for C++ with C++11. C++ has been reworked and is still being

reworked. With C++11, C++14, C++17, C++20, and the newest C++23, it has become possible to program in C++ in such a way that you can write more comprehensible, more error-free, and more sustainable programs—if, of course, you use the elements correctly.

But what makes this *new* or *modern* C++ programming?

- You can program more compactly because the compiler can relieve you of a lot of ballast. *Type inference* with `auto` and the range-based `for` are examples of this.
- You will write safer code if you prefer *unified initialization* with `{ ... }`, use fewer *raw pointers*, use the *containers* and *algorithms* of the standard library, and, last but not least, internalize RAII (see [Chapter 17](#), the fourth of the “dan” chapters on good code).
- By *moving* instead of copying, you become more efficient without having to do anything except leave things out (see [Chapter 17](#)).
- New C++ constructs are an alternative to less secure C means.

When I present examples in this book, I want them to be instructive and meaningful, yet simple, short, and clear in book form. Practical relevance is very important to me. And it can happen that I use one or two *patterns* that might not have been necessary for the example at hand. For example, you will come across the *Pimpl pattern* in [Chapter 11](#). In contrast to patterns such as RAII, which are more important in the modern C++, the explanation of these other patterns will usually be brief, because here my consideration tends toward the short. All established patterns and idioms also belong in a book, but this book focuses on the modern ones.

1.2 “Dan” Chapters

At strategically appropriate points, I have included special chapters that deal less with C++ per se, but will still help you when programming with C++. The topics are of a general software development nature such as modularization, testing, and resource management, but applied specifically in the context of C++.

These “dan” chapters are located at points in the book that are thematically related to the surrounding chapters. However, they are deliberately broad in scope and do not just build on the previous chapters. They have more of a general handbook character and invite you to pick up practical tips later on. When working through the chapters sequentially, a bit of jumping ahead is acceptable and even encouraged.

Special attention should be paid to [Chapter 29](#), which, although very compact, contains the most practical tips.

1.3 Presentation in This Book

In this book, I consistently use C++11, C++14, C++17, and C++20 as standards when explaining topics to you. Most new C++ compilers support the majority of these features. I highlight most C++20 features, because you may not be able to use the latest compiler in your project. If you're working in an environment that requires a very old compiler, you may have to forgo some useful features from the C++ language core and only have access to a limited part of the standard library. Consult your old compiler's documentation to see what you can use, and check *boost* to see if your standard library can be extended with it.

I will mention things that you will only find in the latest C++ version, C++23, in the text and mark them in the source code.

1.4 Formatting Used

Listings contain the following elements:

```
// https://godbolt.org/z/jrqEGvh1M
#include <iostream>           // cout
#include <memory>             // make_shared
int main() {                 // a comment
    std::cout << "Blopp\n";  // highlighted
    type feh-ler(args);      // ✗ line with an error
    if consteval {           // point out C++23 features
        sin(55); }
    for(;;) break;          // other marker, for distinction or highlighting
}
```

Listing 1.1 A small formatting example.

Boxes

Boxes contain important things that you should remember.

Bars

The insertions marked with a bar usually contain further information. At the beginning of most chapters, you will find a *chapter telegram* with terms introduced with this marking.

When I use a name for a sequence of symbols in the text, I try to include the sequence of symbols directly after it. I sometimes would have found this helpful when reading other books. I don't bracket the symbol with quotation marks "" because in many cases I find this cluttering and sometimes even confusing. Here is an example: you write strings in double quotation marks ", use a reference &, use round brackets (), and angle brackets <> and put a comma , between parameters.

Sample Material for Download

You can download program examples and listings at www.rheinwerk-computing.com/5927.

If you have any questions or comments, or if a discussion would help you, you can reach me at <http://cpp11.generisch.de> or by email at torsten.t.will+en@gmail.com.

1.5 Let's Talk Lingo

In the German edition of this book, there was an explanation here about why I prefer English terms over German ones in many cases. We don't have that issue in this edition. However, we have a different one: One reason that I prefer to use English technical terms in the German text is to highlight them as C++ language constructs. So I can write "Konzept" when I mean a general principle and "concept" when I refer to C++ concepts. To maintain this clarity in this edition, I need alternative formulations. I hope I have succeeded in this.

For some things, in addition, it is necessary to define terms. Especially in the context of C++-specific terms, I try to stick to one term for this book when several terms are common for one thing. Some mean the same thing, and for others it is important not to confuse them with counterparts. Here is a short list of important aliases that you might find elsewhere:

- *Destructor* luckily has none.
- *Copy constructor* can be abbreviated to copy-c'tor and means `T(const T&)`.
- *Move constructor* might be a movator or `T(T&&)`.
- *Assignment operator* or *copy operator* instead of copy-assignment operator, copy-assign operator, or `T::operator=(const T&)`.
- *Move operator* instead of move-assign operator or `T::operator=(T&&)`.

Apart from that, I try above all to use clear language throughout. If I haven't quite struck the perfect balance between clarity and style, I hope you'll forgive me.

Chapter 2

Programming in C++

C++ is a programming language for many purposes. In general, it can be used for almost anything. Due to its focus on performance and interoperability, it is often used in system programming. Operating systems, drivers, and other machine-related programs are often written in C++.

Perhaps you come from Java, C#, or JavaScript. Then you are already familiar with programming. Nevertheless, I would like to introduce you to some special features of C++ that may not be clear to you at first glance:

- **C++ is translated into machine code.**

JavaScript is interpreted, and Java is translated into an intermediate code, which is then interpreted. C++ is translated by the compiler directly into the language spoken by the machine.¹

- **C++ is type safe.**

C is not type safe to this extent, nor is Python or JavaScript.

- **C++ is generic.**

You use templates to write generally valid procedures for several data types. This is more difficult in C. In Java, you have *generics*, but they are not as powerful. The prototype-based class concept of JavaScript and Python allows similar results but takes a different approach.

- **C++ allows metaprogramming.**

You can write programs that are executed at compile time.

- **C++ is an ISO standard.**

This means that a global international committee decides on the language. Java is mainly backed by Oracle, Python by the “community”.

The most important difference, however, is that in C++ you *can* choose the best of different paradigms to achieve your goal. However, you do not *have* to follow any of the paradigms. If you want object orientation through inheritance and dynamic polymorphism as in Java, this is also available in C++, but you can also program without it. If you prefer static coupling via *traits*, which could be informally called *static polymorphism*, this is also possible in C++ and is used in abundance in the standard library, but you can also make full use of *virtual methods* instead. If you like functional approaches, you are

¹ At least, that is the usual way. The standard does not prescribe this, and there are C++ variants that do it differently.

better off with C++ with functors, lambdas, and pipelines on views than with Java, but you can also do without them. You can even shift a lot of the execution time to the compile time and thus save users waiting time by assigning more tasks to the compiler with *metaprogramming*—but you can also do without this if it is not important to you.

2.1 Compiling

When you write a C++ program, this means that you write *source code* as text, which C++ tools translate into an executable program. When we talk about these tools, we often refer to the *compiler*, but in reality we mean several tools. I want to be precise in this section and tell you what the different tools do. Later, I will group them together again under the not quite correct term *compiler*.

One of the reasons for this unclear naming is that nowadays the tools are rarely separate programs. They are often just *phases* of a single program. And indeed, [Figure 2.1](#) only reflects a simplified process.

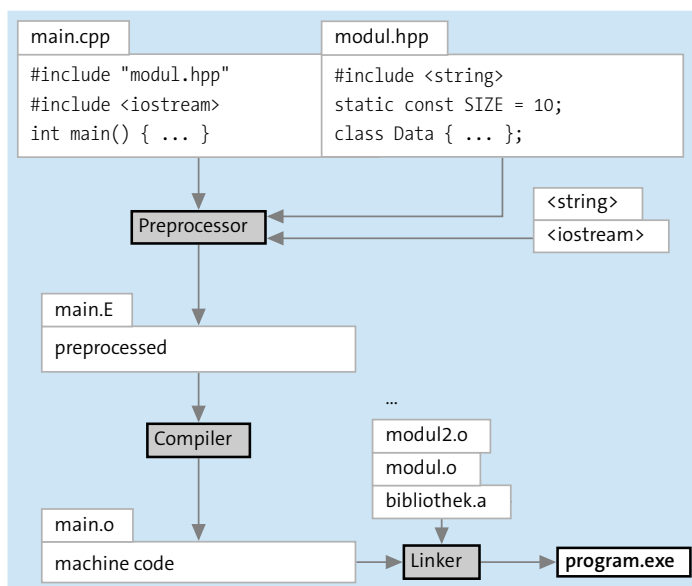


Figure 2.1 The phases of compilation from source code to executable program.

I have left out the optimizations and the fact that a program uses additional libraries (dynamic libraries) when you run it.

This entire process is either initiated from the *integrated development environment* (IDE) or executed manually on the command line. *Manually* is actually an exaggeration here as very often a tool is also used in this process, called the *build tool*. It is quite common to have *Makefiles*. These are text files that contain the components that belong to the program and instructions for how to produce them from your source code.

Later in the chapter, we will build a small program with an IDE as well as execute it on the command line using a Makefile.

2.2 Translation Phases

To ultimately turn the source code you write into an executable program, the compiler suite performs several phases. Nowadays, the suite is often just one program that takes care of everything. It may also be divided into frontend and backend pieces. In the past, the phases were more or less executed by separate programs. The most important phases are as follows:

- **Preprocessor**
Reads the include files and expands C macros—that is, textually inserts a previous definition instead of the macro name
- **Lexer and parser**
Translates the character strings into *tokens*—that is, groups characters into semantic units and recognizes, for example, whether a unit is a string, a number, or a function
- **C++ semantics**
Turns templates into real functions and classes and classes into their instances
- **Intermediate representation (IR)**
Creates a machine-oriented, yet platform-independent code and optimizes it at a high level
- **Code generation**
Generates platform-dependent machine code in assembly language and optimizes at a low level
- **Object files**
Translates the assembly language into machine-readable byte sequences, outputs them to object files and static libraries, and adds debug information
- **Link**
Creates an executable file or dynamic library from object files and library files

2.3 Current Compilers

As already mentioned, the term *compiler* does not cover a single translation program, but the entire tool chain from the preprocessor to the linker. In addition, the *standard library* is an integral part of C++. So if you install a compiler on your system, you will always receive a standard library with it.

With well-meaning intentions, this book largely refers to the current standard known as C++23. For C++20, the majority is implemented in the current compilers. I emphasize

things that belong to C++23 because the language support is still incomplete, but it is improving. If you use the possibilities that have been added since C++11, you will be able to learn the language faster and use it better than was the case with the previous versions up to C++98, for example.

With a modern compiler, C++ is definitely more enjoyable. Fortunately, most compilers are more or less up to date.

2.3.1 Gnu C++

The C++ compiler *g++* from the *Gnu Compiler Collection* (GCC) is the compiler available on most platforms. It is also the testing ground for trying out new things, so you will almost always find new features implemented here first. Today, in the middle of 2024, C++20 is very well implemented. Unfortunately, C++23 still lacks some features, especially for ranges. On Linux, GCC is usually the first choice. Although GCC is widely used, it has a reputation for having a very complex code base. The compiled programs fall behind somewhat in terms of speed compared to paid compilers.

2.3.2 Clang++ from LLVM

As far as the code base is concerned, *LLVM* with the C++ compiler called *Clang++* has a better reputation. The implementation of the C++20 features is exemplary; little is missing from C++23. Some new features are implemented here first. Clang++ is the standard compiler for macOS development. It is available free of charge for Linux but must be installed in addition to an existing standard library, so it is best to install *g++* beforehand.

2.3.3 Microsoft Visual C++

Microsoft's product world for C++ consists (together with other tools) of *Microsoft Visual C++* (MSVC), the standard library *MSVC STL*, and the IDEs that integrate them. These are the specialized *Microsoft Visual Studio*, which is only available for Windows, and the cross-platform *Visual Studio Code*. MSVC and MSVC STL differ only slightly from other implementations in terms of the implementation of the C++20 standard. Most of the most important features are included. In some C++23 features, this compiler is even ahead of the others.

2.3.4 Compiler in the Container

With a suitable Docker container, you can try out other compilers with your code without installing them on your system and potentially messing it up.

GCC offers a whole range of ready-to-use containers. For example, you can compile a piece of source code with g++ in version 14.1, which already supports some of C++23 (https://hub.docker.com/_/gcc):

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir gcc:14.1 \
  g++ -std=c++23 -Wall -Wextra -pedantic myFile.cpp -o myFile.x
```

The result can usually be executed directly with `./myfile.x`. Sometimes there are dynamic libraries in the container, in which case you must also execute the file in it:

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir gcc:14.1 \
  ./myfile.x
```

With Clang++, it works the same way, except that you use `silkeh/clang:18` as the image for this compiler.

2.4 Development Environments

There are two main ways for you to develop C++ programs:

- **Command line**

You work on the command line and call the compiler and other tools manually. Later, you will use tools such as Makefiles to automate these tasks. I recommend that you at least try out the command line. On the one hand, you also learn other useful things about programs and programming. On the other hand, it is easier to automate processes on the command line—and that is ultimately what programming is all about.

- **Integrated development environment**

An IDE that is tailored to your personal needs can increase your productivity immensely, especially later on in your everyday programming life. On the other hand, an IDE can also overwhelm a beginner with its flood of features. There are assistants that try to speed up the introduction. Whether this works depends on your personality. If you are completely unfamiliar with the command line, you can give this a try.

In some cases, the choice of compiler determines the IDE. If you opt for Microsoft, you have a choice of IDE between *Visual Studio Code* and *Visual C++*. Visual Studio Code is free, runs on several platforms, and is intended for various languages. For C++, simply install the corresponding plug-in. Visual Studio C++ only runs on Windows, but it specializes in C++. The website currently advertises the 2022 edition. The already very comprehensive basic version, *Visual Studio Community*, is free of charge. The *professional* and *enterprise* editions are subject to a charge.

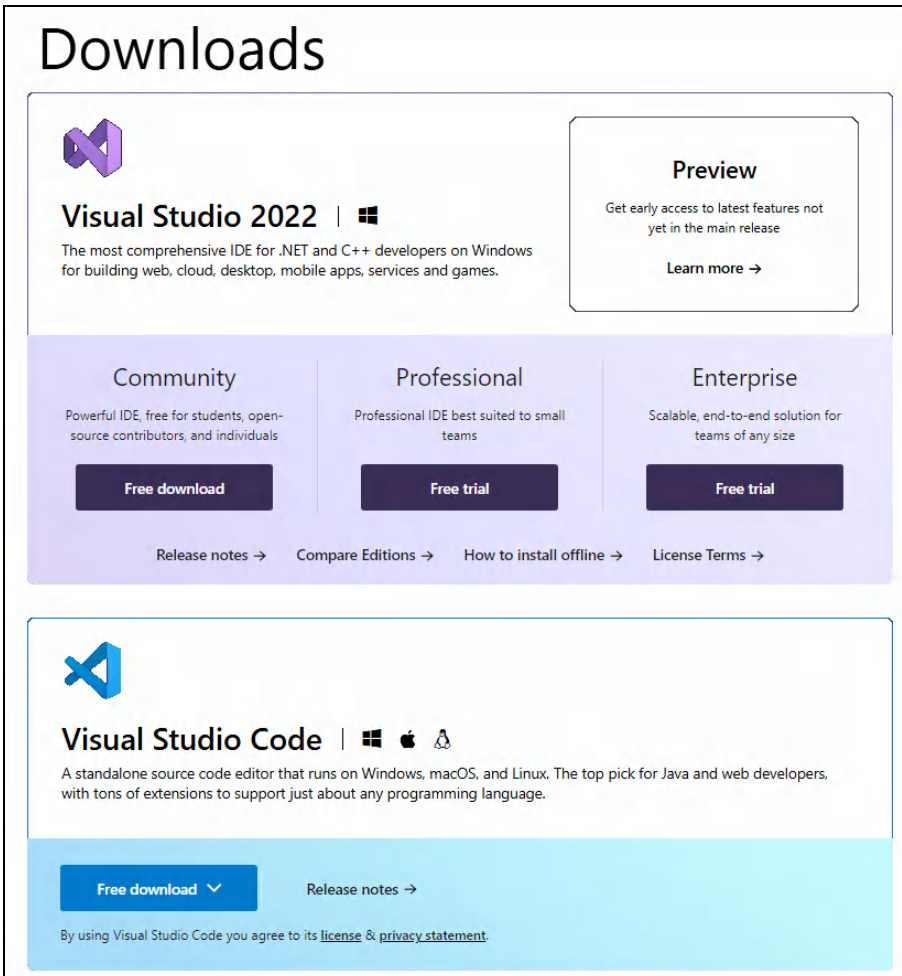


Figure 2.2 Microsoft products for C++ development.

On the Mac, Xcode supplied by Apple is the de facto standard. This gives you the choice between an excellent IDE and a collection of tools for the command line. You can download these from Apple (<https://developer.apple.com/xcode>). You can currently obtain Xcode 15 with compiler version 15. The compiler without an IDE can also be used under Linux, where the latest version 18 is available (<https://releases.lldvm.org/>).

The GCC is available as an alternative for Unix as well as for Windows and on the Mac. The C++ compiler is called *g++*—currently in version 14.1. You primarily use it from the command line, but it also integrates into IDEs such as Eclipse CDT, NetBeans, KDevelop, Code::Blocks, Qt Creator, and others. Some of these tools are even available for multiple platforms. If you want to develop with *g++* on Windows, then look for the MinGW integration and whether you have to download it separately or whether it is already included (*Minimal Gnu for Windows*).

Linux Subsystem under Windows

Under Windows, there is an optional *Windows Subsystem for Linux* (WSL). There you can start a bash shell and have the same commands available as under Ubuntu, for example—including the `apt-get` installation command. You can then install GCC and many other things under Windows.

A thoroughly commercial but interesting solution is *CLion* from JetBrains. The IDE is very well thought out and is based on the successful IntelliJ IDEA, which makes it interesting for some Java developers. Its own compiler is a little behind in terms of standards, but you can configure an installed GCC or Clang compiler.

2.5 The Command Line under Ubuntu

As an example for the development with the command line, I will give you a quick guide for the current long-term version of Ubuntu, a widely used Linux distribution. If you are using a different Linux distribution, the commands may differ.

You install `g++` and some useful tools like this:

```
sudo apt-get install g++ make
```

You enter the program code in an editor, and this is where the real agony of choice begins. If you are using an IDE, the editor is included. Without an IDE, you enter your program in any general text editor. Text editors are a dime a dozen.

I am only suggesting two that meet different requirements: *gedit* and *kate*. The choice between *gedit* and *kate* should depend on whether you are using Gnome or KDE as your desktop. Simply try out which of the following two commands would install fewer packages automatically, and then select the editor:

```
sudo apt-get install gedit
sudo apt-get install kate
```

Between the pure text editors and the fully comprehensive IDEs are the new products *Visual Studio Code* from Microsoft and *Fleet* from JetBrains, the latter still as a preview. Both run on all platforms and are very popular.

If you join a team with several developers, find out whether *Emacs* or *Vim* will be used. These are very popular text editors among programmers, but they have a steep learning curve. If you have colleagues who can help you get started, feel free to choose one of these two editors:

```
sudo apt-get install emacs
sudo apt-get install vim
```

2.5.1 Create a Program

We will come to the IDE in a moment when we discuss Visual Studio Code under Windows as an example. Now let's walk the path.

Open a command line, sometimes also called a *terminal* or *console*. You will certainly find an entry for this in the menu. In Ubuntu with Gnome, you can also press `[Ctrl]+[Alt]+[T]`. A new window should open with a blinking cursor showing a command line similar to this one, the so-called prompt:

```
towi@havaloc:~$
```

In the rest of this book, I will not mention `towi@havaloc:`, which stands for user and computer names, or the tilde `~` for the current working directory. The prompt `$` means that you should enter a command after it. Practice creating a new directory and entering it:

```
~$ mkdir quellcode
~$ cd quellcode
```

Your prompt should now contain the directory to which you have changed:

```
~/sourcecode$
```

Because there are so many Linux flavors, it is quite possible that your display will look different even though you have done everything correctly. You can use `pwd` to check which directory you are currently in.

Now open the editor of your choice. You can do this via the menus or enter the name of the file you want to edit on the command line. Add an ampersand `&` so that you can continue typing even though the editor is open (if you forget this, press `[Ctrl]+[Z]` in the command line and then type the command `bg`, followed by `[Enter]`). I myself am an Emacs user, but use your favorite editor here:

```
$ emacs modern101.cpp &
```

Type the following source code into the editor window. You should recognize somewhere that you are really editing `modern101.cpp`.

```
// https://godbolt.org/z/911fx3nq
// modern101.cpp : Fibonacci console
#include <iostream>
int fib(int n) {
    return n<=1 ? n : fib(n-2) + fib(n-1);
}
int main() {
    std::cout << "Which Fibonacci number? ";
    int n = 0;
```

```
std::cin >> n;
std::cout << "fib(" << n << ")=" << fib(n) << "\n";
}
```

Listing 2.1 Each Fibonacci number is the sum of the two numbers before it.

Translate this source code into the executable program `modern101.x`:

```
$ g++ modern101.cpp -o modern101.x
```

Here `g++` is the compiler. Use `modern101.cpp` to specify the source file. If you have several source files that you want to compile into a program, specify several `*.cpp` files here. Use `-o modern101.x` to tell this compiler the desired output file name. If you forget this, it doesn't matter; the finished program will end up in `g++` in `a.out`.

Try it out for yourself:

```
$ ./modern101.x
Which Fibonacci number? 33
fib(33)=3524578
```

This is your first C++ program. Congratulations!

By the way: Under Windows, executable programs are normally given the extension `*.exe`. Under Linux, an extension for executable C++ programs is rather unusual. To illustrate this, I create Linux programs here, but with the extension `*.x`—a practice that I also use from time to time in the “real world”. Whether you follow my example or not is up to you.

If you are interested in how you can speed up the program, move ahead to [Section 2.7](#).

2.5.2 Automate with Makefile

However, as it is tedious to call the compiler again and again in this way, it is best to create a Makefile in which the necessary commands are listed. To do this, create the following Makefile file:

```
$ emacs Makefile &
```

The content of the file is then simple:

```
# -*- Makefile -*-
all: modern101.x
modern101.x: modern101.cpp
    [Tab] g++ modern101.cpp -o modern101.x
# clean up:
clean:
    [Tab] rm -f *.x *.o
```

The comment lines with # are not essential. Make sure that the indented lines do not start with a space but with a *tabulator*, which I indicate here with [Tab]. I won't go into all the details here, but the following two lines,

```
modern101.x: modern101.cpp
[Tab] g++ modern101.cpp -o modern101.x,
```

say to make: “If you need to create modern101.x, then you need modern101.cpp; to create it, run the command `g++ modern101.cpp -o modern101.x`”.

If you now run

```
$ make
```

then the `all:` rule will look up what you want to have built. You can list other rules there, which `make` will then execute one after the other. You should now have your program built again. `make` may notice that nothing has changed, in which case a `make clean` (execute cleanup rule) or `make -B` (pretend everything has changed) will help.

You can also use `make all`, `make modern101.x`, or `make clean` to execute one of the other rules. Isn't that convenient?

2.6 The Visual Studio Code IDE under Windows

To install Visual Studio Code (VS Code), simply type “vscode download” in the search bar of your browser and then click the download link that appears. Then download the setup and run it.

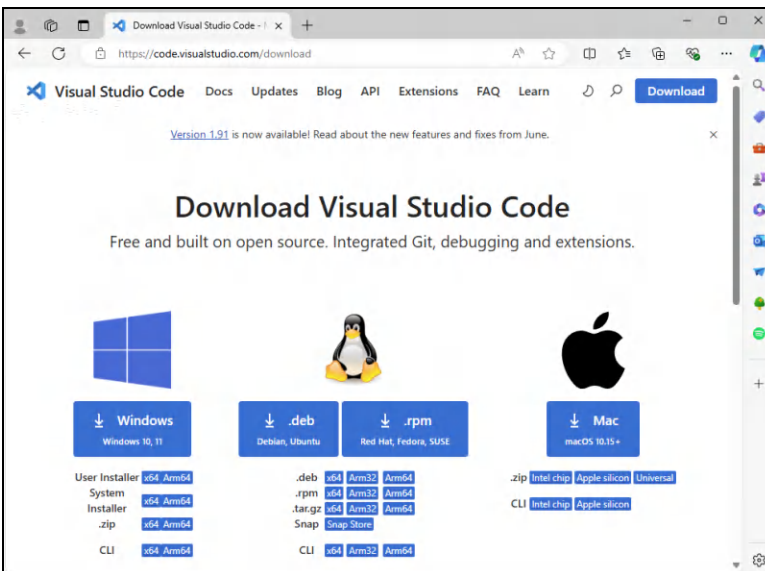


Figure 2.3 Visual Studio Code is available for popular operating systems.

You'll then find the following:

- License agreements
- An installation location that must be selected
- The start menu folder to set
- Options such as **Add to PATH** (available after restart)

You can then start VS Code in the last installation step or simply type “code” and press Enter in the Windows search bar.

Visual Studio Code under Linux

VS Code also runs excellently under Linux. Perhaps even a little better as the compiler is typically already installed there and available in the path. Download VS Code from the website and install it. Also install the C/C++ extension. The compiler available in the system will be recognized.

The display language is (probably) initially set to English. If you want to have the IDE use a different language, you can install that language later by pressing Ctrl + Shift + P and selecting **Configure Display Language**.

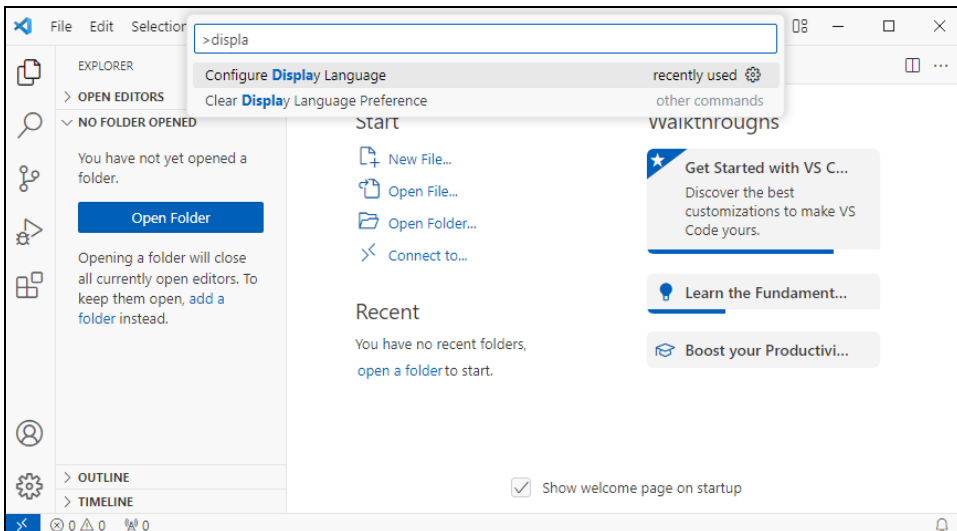


Figure 2.4 You can first change the display language.

If you choose to change the display language, VS Code will restart with the menus in the new language. It is a good idea to follow the installer's recommendation and also to restart Windows once.

Next, install the *C/C++ plug-in*:

- Pressing `Ctrl` + `Shift` + `X` or clicking the four-box icon in the left bar opens the extensions.
- Enter “C++” in the search and the original Microsoft C++ extension will appear in the list. You can choose the pure extension or the extension pack. I myself have opted for the latter. It consists of the actual C++ extension, the CMake extension, and C++ themes.

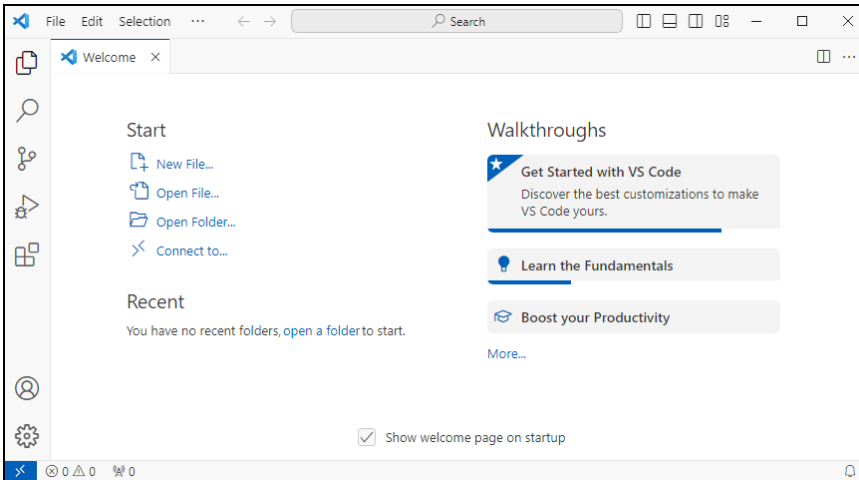


Figure 2.5 The welcome screen.

Now we still need a C++ compiler. Microsoft recommends that you check whether you already have one installed. We assume that this is not the case. You have a choice between the *MinGW g++* or *MSVC* compiler.

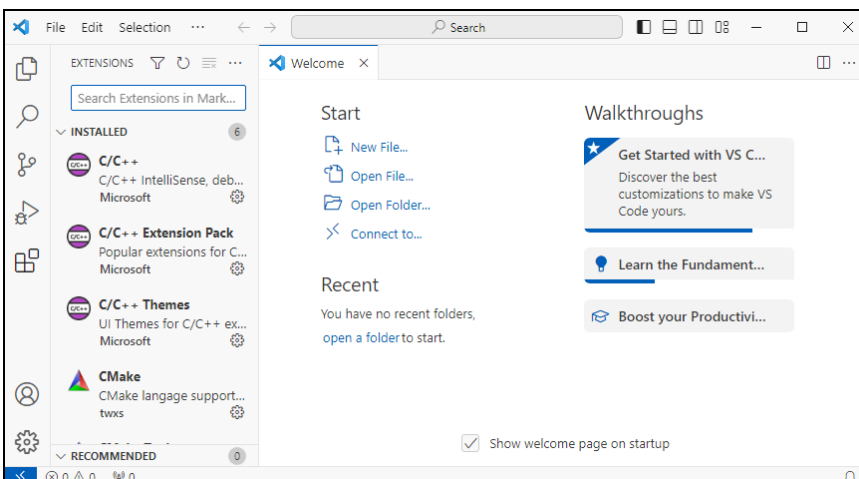


Figure 2.6 Programming languages are supported in VS Code with plug-ins.

We use the MSVC compiler here. It is best to close VS Code. Then download **Build Tools for Visual Studio 2022** from <https://visualstudio.microsoft.com/downloads/> and install them. In the selection that appears, select **Desktop Development with C++** on the left-hand side and at least **MSVC** and **C++-CMake** on the right-hand side.

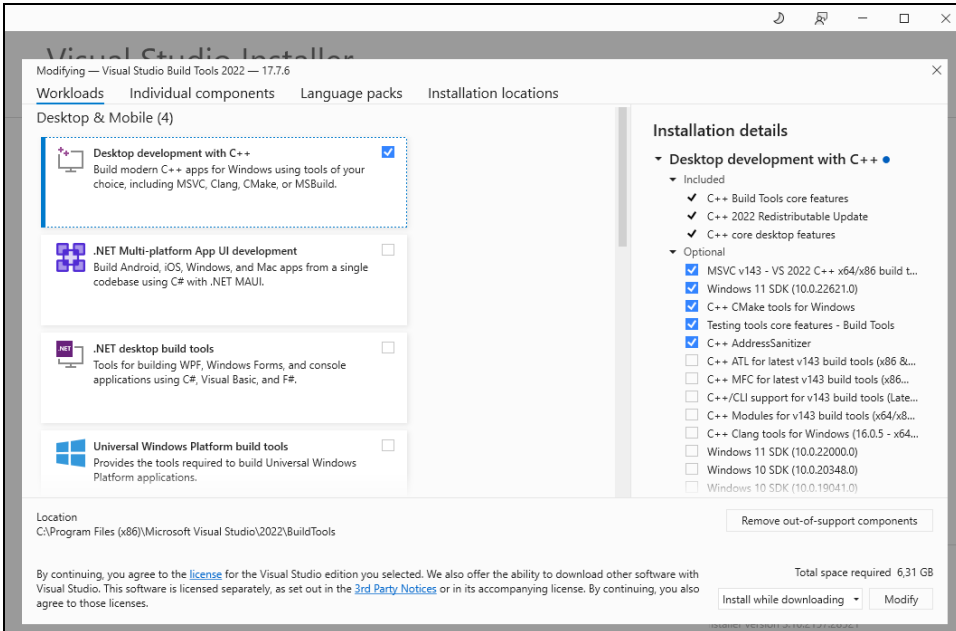


Figure 2.7 The compiler and other tools are installed separately.

When the many packages have been installed, click **Start** in the installation screen and you will land in the **Developer Command Prompt for VS 2022** (the Devconsole). You can close the installer. You can open the Devconsole again and again by entering “Developer Command” in the start bar and selecting **Devconsole**.

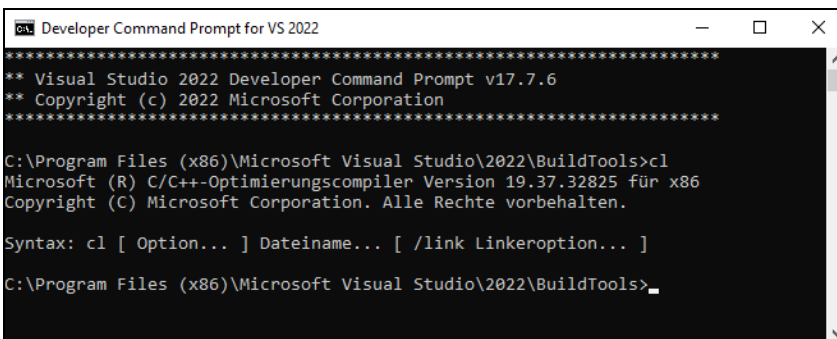


Figure 2.8 The cl compiler is available in the Devconsole.

To start VS Code with the installed compiler, enter “code” in the Devconsole.

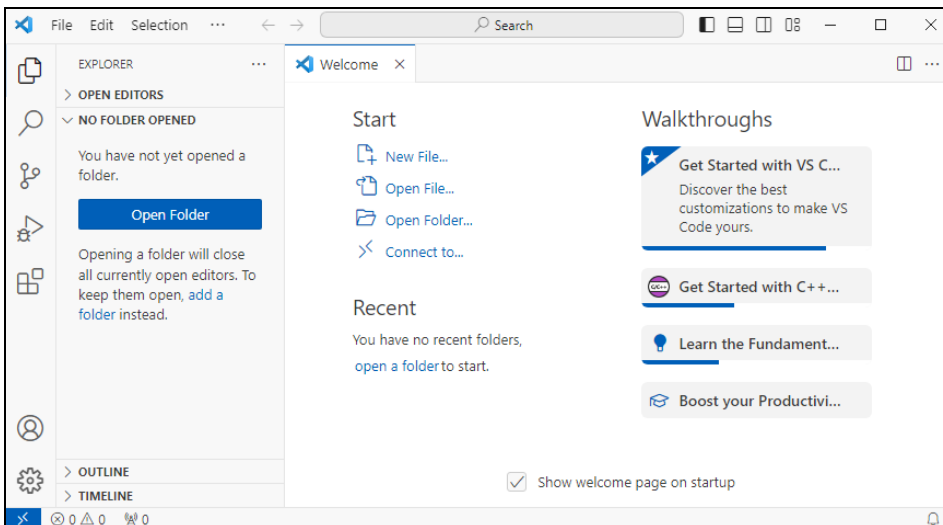


Figure 2.9 The fully installed IDE with C++ support.

You will see that the C++ plug-in welcomes you with a new button. Click it. After a few seconds, you can click the button labeled **Select My Default Compiler**. There is actually already a blue **Done** tick there, but you can easily check the setting.

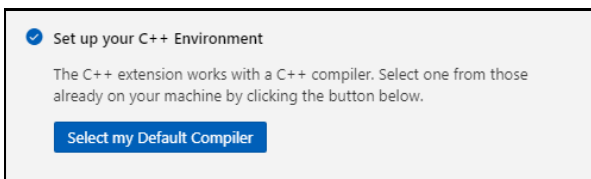


Figure 2.10 The compiler has been found.

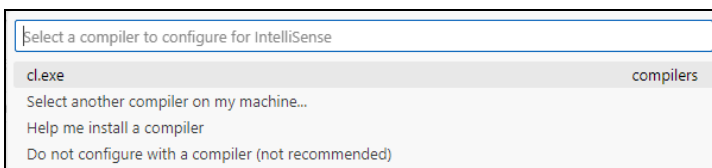


Figure 2.11 Set the default compiler.

Then select **cl.exe**. This is the compiler that you have already tried out in the DevCon-sole. You can carry out the further steps of **Get Started with C++ Development** yourself if you wish. However, I can also show you a different or additional way to a demo C++ program.

Open a terminal with `Ctrl`+`0` in VS Code. You should get a command prompt in your home directory. There, enter `mkdir modern101` and `cd modern101` in succession to change to a new directory.

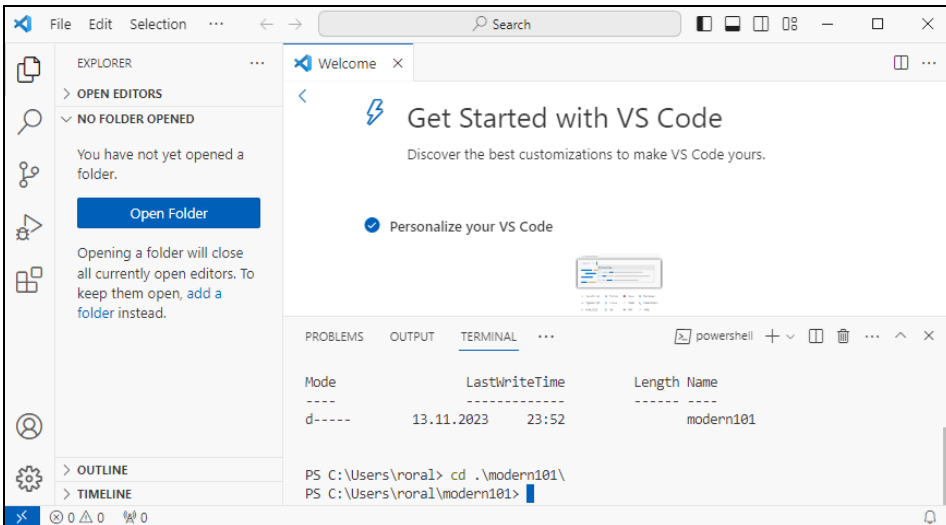


Figure 2.12 Start a project in a new directory.

Then select the clearly visible **Open Folder** button in VS Code and select the **modern101** folder you have just created. A new project window will open in which you must first agree to the sentence **Yes, I trust the authors of this folder** (that's you).

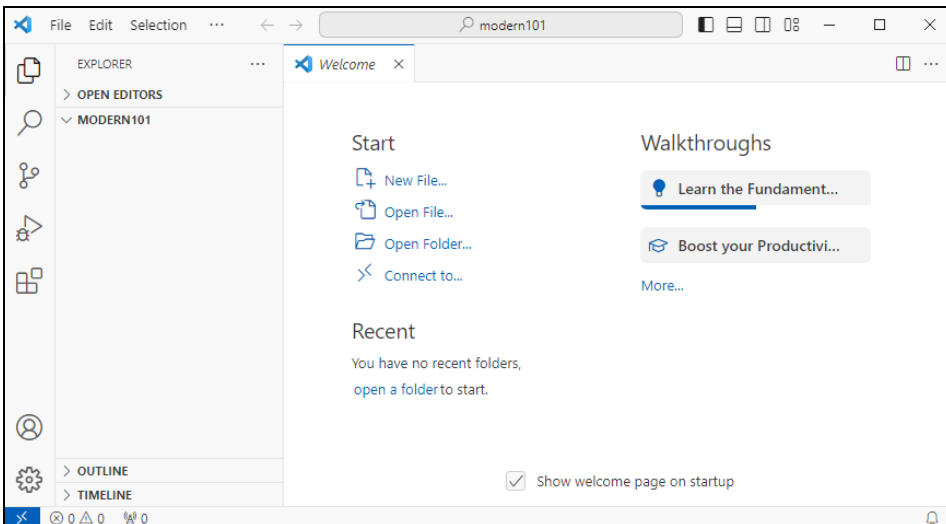


Figure 2.13 A new VS Code opens with the fresh project.

You will now see the project area on the left. There you can hover your mouse pointer over **MODERN101** to display buttons for creating a new file. You can also select the **New File...** entry in the welcome area under **Start**, but one more dialog will appear. The file name should be **modern101.cpp**. A text editor opens in the right-hand area where you can enter source code. Write your first C++ program here!

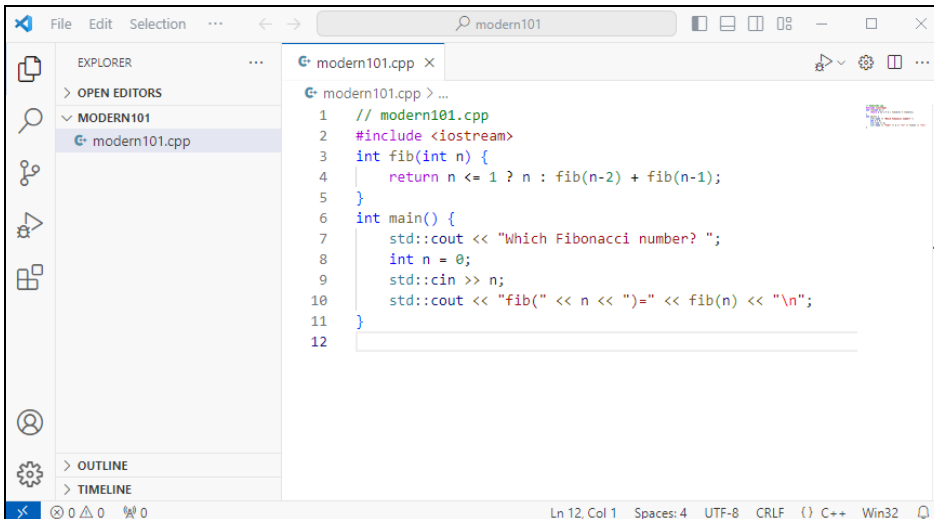


Figure 2.14 Enter the C++ source code in the editor.

Now you only need to click the triangle in the top-right-hand corner and select **C/C++: cl.exe build and debug active file**.

A terminal should open in VS Code, in which the program is compiled. Eventually you will see the question, **Which Fibonacci number?** To test it, enter "45".

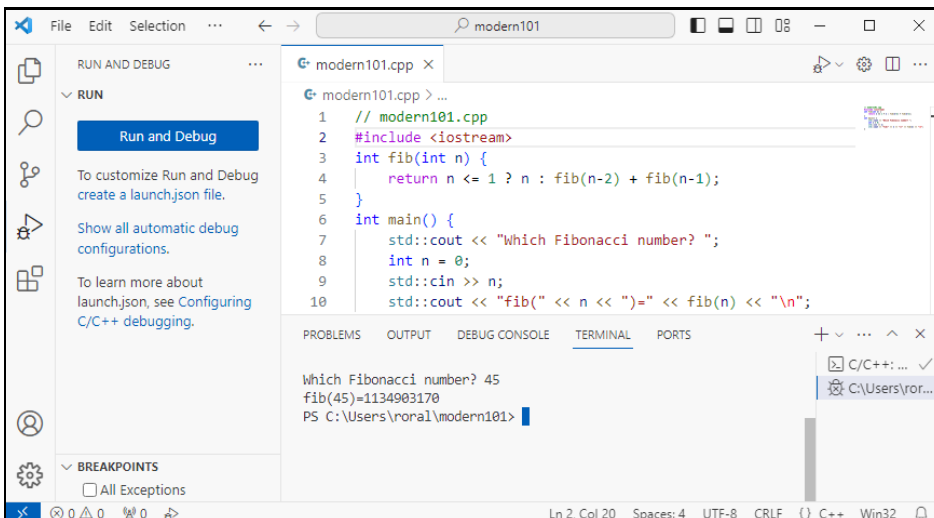


Figure 2.15 Your first executed C++ program.

With the number 45, it takes a while, but then you will see the result in the terminal. On my computer, the calculation takes about a minute. You can also start with a smaller number like 20. If you choose a number larger than 45, you will exceed the capabilities of the program and get nonsensical output.

Note that the number ranges refer to a Win32 application on a 64-bit Windows. The limits may be different on other platforms.

2.7 Speed Up the Sample Program

If the program runs too long for you, then *tabulate* the intermediate results. This means that you save them in a table-like data structure. The zeroth entry receives the result of `fib(0)`, the first the result of `fib(1)`, and so on. If you want to use `fib(n)` to calculate the *n*th entry in the table, do not call `fib(n-1)` and `fib(n-2)`, but look in the table instead. Then you will get the results faster than you can look. Create a new project or modify the program:

```
// https://godbolt.org/z/3W1qY7b9x
// modern102.cpp : Fibonacci console
#include <iostream>
#include <map>
int fib(int n) {
    static std::map<int, int> table{};
    table[n] = n<=1 ? n : table[n-2] + table[n-1];
    return table[n];
}
int main() {
    std::cout << "How many Fibonacci numbers? ";
    int n = 0;
    std::cin >> n;
    for (int i = 1; i <= n; ++i)
        std::cout << "fib(" << i << ")=" << fib(i) << "\n";
}
```

Listing 2.2 A quickly created table of Fibonacci numbers.

If you enter “50” here, for example, then you will see that the results are sometimes negative from 46—a sign of an overflow. The same applies here: On platforms other than 32-/64-bit Windows, you may have different limits. This means that the numbers are too large for this program and the program returns nonsense. Having an overflow in a program is usually not a good idea. Therefore, in [Chapter 4, Section 4.4](#), you will learn what to look out for and how to avoid this problem.

Chapter 3

C++ for Newcomers

This chapter is primarily aimed at those transitioning from Java or C#, but newcomers from other higher-level languages, especially object-oriented languages, will also benefit, as will those refreshing their C++ skills after some time. I provide a general overview of the idiosyncrasies of C++ that may surprise or pose difficulties for newcomers.

All other readers can skip this chapter; the overview and introduction follow.

The following elements are ones that many developers should recognize:

- **Statements**

Programs are executed statement by statement, one after the other—at least per thread, at least in the model. The rule of thumb is that semicolons separate statements from each other. Statements can be combined into blocks.

- **Expressions**

An expression recursively consists of expressions down to indivisible units such as literals or variables. Arithmetic expressions contain mathematical calculations, for example. In C++, an exact type can be assigned to every expression.

- **Data types**

C++ offers a range of simple data types such as `int` and `double`. There are also pointers and references, which are separate types in C++. An `int` and a pointer to it `int*` are different types. You can aggregate several types together to obtain new types.

- **Functions and methods**

To prevent programs from turning into long spaghetti, reused areas can be outsourced to functions. Functions that are in a data type are called *methods*.

- **Classes**

Data types that you also bundle with behavior—that is, to which you add methods—are called *classes*.

- **Function calls**

A function call takes parameters and returns a result. What happens within the function is partially invisible from the outside.

- **Parameters**

Functions receive parameters. In C++, the function decides whether the parameter is to be used as a value (*by value*) or as a reference—or pointer (*by reference*)—not the caller.

■ Returns

The same applies to returns from functions. A result can be assigned to a variable or used within an expression. The function decides whether the return value is an independent copy or a reference.

Some things may seem familiar to experienced programmers at first glance, but they have important conceptual differences from other languages in detail. For example, if Java developers bring incorrect preconceptions here, they might later be confused and encounter nasty surprises. Therefore, I want to briefly mention a few things that could be stumbling blocks:

■ Stack and heap instead of garbage collection

It will come as no surprise when I tell you that there is no automatic cleaning up of objects in C++. Don't see this as a disadvantage; live the advantage. Separate between things that are automatically managed on the *stack* by the compiler when the block is exited and those that you request on the *heap* with `new` and for which you take responsibility for clearing away. Don't mess around: better use *RAII* (see [Chapter 17](#)).

■ Virtual versus real machines

It is known that Java code, once translated to `class` files (thanks to the *Java virtual machine [JVM]*), runs on all platforms (“write once, run anywhere”). C++ code must be compiled separately for each platform because the compiler output is directly executable machine code (“write once, compile anywhere”). Although this is not a requirement according to the standard, it is usually the case.

■ C++-char versus Java-char

C++-char is usually eight bits wide and appears on different systems sometimes as signed and sometimes as unsigned. Without the corresponding designation, you can therefore only rely on a value range from 0 to 127. Only if you write `signed char` does it correspond to the Java byte. Java-char corresponds more to C++-short and guaranteed to `int16_t`. The latter does not have to be present, but it is *de facto*.

■ C++-optional versus Java-optional

In Java, you often use `optional` as part of the Java Stream API. In C++, the containers and the new ranges correspond most closely to the Java stream API. However, `optional` is not a container in C++, and therefore you will not use `optional` as in Java. However, with C++23 the *monadic transformations* have been added, which move `optional` in this direction.

■ Function objects versus lambdas

At first glance, Java lambdas are similar to C++ lambdas. On closer inspection, the anonymous function objects in C++ are more rounded. In Java, some effort still has to be made at call time in the JVM to dynamically create a function object. In C++, the compiler has completely outsourced the function and only given it an invisible name. Binding to local variables is similar, but in C++ you can choose between binding as a value or as a reference.

■ Values and references in C++ versus Java and the like

Everything that is an object is a reference in Java. In C++, everything is a value and is copied for parameters and returns. Only with special provision with `&` and `*` can a function explicitly request references and pointers instead.

■ Throwing values instead of pointers

In C++, you do not write “throw new X(...)”, but only `throw X(...)`. If you don't pay attention to this, you will run into trouble in the medium term. For example, you would have to delete the exception objects yourself. And this is not always possible, as is the case with `catch(...)` (*catch-all*). And what about rethrowing? You should instead rely on the C++ mechanisms, where it is guaranteed that an exception instance exists as long as it is needed. The time to deviate from this rule is when a framework requires you to explicitly clear away because it throws exception instances as pointers. In this case, you should of course adhere to what the framework prescribes. Read its documentation to find out whether you should clear the exception in the catch block.

■ `const` versus `final`

Because all objects in Java are references, `final` in Java only refers to the reference marked with it. You certainly know that you can change its content wildly despite `final` if the interface allows this (which fortunately is not the case with things like `Integer`). Because everything is initially a value in C++, `const` also protects the content. In connection with references and pointers, you have even more control, as you can see in [Listing 3.6](#) and its explanation.

■ Templates versus generics

Both use angle brackets, and yet they are completely different. In Java, only a single function or class is created per generic; you are only relieved of the type conversions—for example, for returning a value from methods. The only thing you know about the type parameter is that it is either an object or that it implements a specific interface. A generic therefore always applies to a specific group of objects. In C++, a template is instead a stencil, which must just be parseable C++ code from the compiler, nothing more. Only when you use it do you specify the types of parameters, and C++ then inserts them—and generates the actual function at this moment (at compile time). This can then be different functions for each type.

■ Interface approaches in C++ and Java

In C++, there is *multiple inheritance* without restriction. And because, according to Terry Pratchett, nothing good ever follows the word *multiple*,¹ the Java thinkers didn't want to bring this specific devil into their house either. But you can't do without it completely, because then object orientation would be impossible. This is why Java has implementation-free² interface declarations. This sensible approach puts a

1 *Guards! Guards!*, Terry Pratchett, “The noun doesn't matter after an adjective like multiple, nothing good ever follows multiple.”

2 Ignoring default implementations of Java 8 for once.

stop to the misuse of multiple inheritance and makes it more difficult to design overcomplicated interface hierarchies. In C++, it is the other way around: restrictions that are intended to protect against design errors must come from the developers; they do not come from the language. As a sensible restriction, you can start by adopting the idea from Java: for example, *if* multiple inheritance is used, then make sure that at most one of the parent classes contributes implementations and the rest are only so-called signature classes—that is, classes that contain only *pure virtual methods*. This is what C++ calls *abstract methods*—that is, *virtual* methods that have zero assigned instead of an implementation.

■ Standard library and JEE

If you install Java completely, you get a huge API with it, with libraries that cover pretty much all areas of life. This is only the case to a limited extent in C++. Although the standard library gives you a solid foundation, you are dependent on third-party providers for many important things. This has become increasingly rare over the course of the last language versions from C++11 to C++23, as more and more general-purpose tools have become part of the standard. However, network communication and many other things are missing from the C++ standard library. For now.

Method Declaration, Method Definition, on Site, Separate, and Abstract

For your acute understanding: in C++, you define a method either directly at its declaration in the class or somewhere else in the sources. In the first case, the definition follows the function header:

```
struct Demo {  
    int x;  
    virtual void func() { x = 12; } // Definition on site  
}
```

For methods with more than a few lines, separate the declaration and definition. The function body is then located outside the class definition, preferably in a different file:

```
struct Demo {  
    int x;  
    virtual void func(); // definition (probably) somewhere else  
}  
...  
void Demo::func() { x = 12; }
```

However, you do not necessarily have to have a method definition. This is only really necessary if you also use the method. The linker usually reports an error when linking the entire program. It is therefore perfectly fine to have a method declaration without a method definition.

However, if you write `= 0` in the method declaration, then this is also its definition. Namely, as a purely virtual—that is, abstract—method:

```
struct Demo {
    virtual void func() = 0; // abstract method
}
```

Here are some examples where Java and C++ differ.

In Java, you need the keyword `new` to create a new instance. The instance is created on the heap.

```
Data data = new Data(5);
Data more = new Data(6);
data = more;
more.value = 7;
System.out.println(data.value); // always 7
```

Listing 3.1 Java: A small Java example with the ubiquitous object references.

`data` and `more` are *references*. In fact, they are more like pointers in C++, because they can also assign the null pointer `null` to `data`, which would not be possible with references in C++. The assignment `data = more` means that `data` and `more` now reference the same instance. In C++, you would rather write:

```
Data data{5};
Data more{6};
data = more;
more.value = 7;
cout << data.value << '\n'; // still 6
```

Listing 3.2 C++: In C++, you have values instead of references.

Here, `data` and `more` are separate values. Assuming that `more{6}` resulted in the initialization `more.value=6`, then `data.value` still contains the 6 even after the assignment of `more.value=7`.

And if you use heap and pointers with `new` in C++ to emulate Java behavior, don't forget that you don't have a *garbage collection*, but have to release the heap again somehow. To do this, either use an auxiliary class such as `shared_ptr` (recommended) or an explicit `delete`:

```
// https://godbolt.org/z/WM11Gs6sz
auto data = make_shared<Data>(5);
auto more = make_shared<Data>(6);
data = more;
more->value = 7;
cout << data->value << '\n'; // now also 7
```

Listing 3.3 C++: Modern C++ with heap memory uses auxiliary classes such as `shared_ptr`.

This is already very “modern,” but you can also make it “purer.” In the next example, you can see the whole thing without the wrapper objects created by `make_shared`.

```
// https://godbolt.org/z/Ee7eqq1MY
Data* dataOwner = new Data(5);
Data* data = dataOwner;
Data* more = new Data(6);
data = more;
more->value = 7;
cout << data->value << '\n'; // now also 7
/* tidy up */
delete more;
delete dataOwner;
```

Listing 3.4 C++: Without modern C++ tools, special attention must be paid to pointer ownership.

But oh dear: here you can already see what trouble you can get into if you have to handle clean up yourself. `delete data` would have been wrong in the end, because `data` points to `more`, and that has already been cleared away. The assignment `data = more` is the villain that has hidden `data`. Release is no longer possible. This is the only reason that `dataOwner` exists, so that the original pointer can be released later.

Therefore, in C++, it is useful to adopt the concept of *ownership of pointers*: raw pointers always belong to someone who is responsible for clearing them away. Either this is the person who created the object with `new`, or the ownership is transferred. You can do this “in your head,” using tools such as the *Guideline Support Library* (GSL; see [Chapter 30](#) on guidelines); write wrapper classes yourself; or simply always use the supplied tools such as `shared_ptr` and `unique_ptr`. In the next example, I choose `final`. In Java, this means that nothing new can be assigned to the reference.

```
final Data data = new Data(5);
data.value = 7;           // that's okay
data = new Data(6);       // ✗ this prevents final
```

Listing 3.5 In Java, `final` only blocks the reference.

In C++, `const` can be used in a much more differentiated way.

```
// https://godbolt.org/z/Moe5M7sK8
Data const * data = new Data(5);
data->value = 7;           // this const protects Data
data = new Data(6);       // reassigning pointer is okay
Data * const more = new Data(8);
```

```
more->value = 9;           // now okay  
more = new Data(10);      // reference is protected
```

Listing 3.6 C++: const can protect the value or the reference.

In C++, both the reference or pointer and the content—that is, the value—can be protected. You can also combine both with `Data const * const`. You can recognize what is protected by the `const` *behind it*:

- **Data * const more**
The pointer or reference is protected.
- **Data const * data**
The value `Data` is protected.
- **const Data * data**
This is an alternative notation for `Data const * data`.
- **Data const * const more**
Both value and pointer are protected.

There are also variants of `const`: `const` methods may not change their own object, and `constexpr` forces the expression to be calculated by the compiler at compile time.

Chapter 4

The Basic Building Blocks of C++

Chapter Telegram

- **main**
The entry point to every program.
- **#include**
Integration of other program parts and libraries.
- **Variable**
Name for a memory area that can hold a value.
- **Initialization**
This is the value that a variable will have when it is created. For built-in types, initialization is particularly important during definition as otherwise a variable will have an undefined state.
- **Assignment**
A special *expression* that changes the content of a variable using =.
- **return**
Exiting a function; in *main*, the end of the program.
- **Comment**
A note to the program code that the compiler does not evaluate.
- **Statement**
A program is the sequential processing of various statements.
- **Expression**
A sequence of operations on operands for assignments and the like.
- **Block**
A group of statements between curly braces.
- **Type or data type**
For the compiler, every expression has a type.
- **Standard library**
Part of the C++ standard and supplied with the compiler. Everything that is not pure syntax or semantics or a built-in data type is supplied by the standard library.
- **Own function**
With a custom function, you outsource code to another location. This is the basis for clarity and reusability.

- **Function parameter**
Gives the “thing” passed to a function its own local name within a function.
- **Side effect operator**
Operator that changes the value of a variable but is not an assignment.
- **int and bool**
Two elementary (built-in) and simple types.
- **Unified initialization**
A unique notation for the first value of a variable or a default parameter.
- **Token**
For the compiler, the smallest building blocks of the program text.
- **Identifier**
Names of program elements—that is, variables, types, functions, and so on.
- **for statement**
This is a way of implementing the repetition of statements.
- **if statement**
You use this to implement branching.
- **Operator**
Usually a symbol that stands between two operands (or in front of one); works like a function with the operands as arguments.
- **Operand**
Argument for an operator.
- **Arithmetic operator**
Used for classical arithmetic with +, -, *, /, and % as well as bitwise arithmetic with |, &, ~, <<, and >>.
- **Relational operator**
Greater than, less than, equal to, combinations thereof and not equal to: >, <, ==, <=, >=, !=, or even <=> (C++20).
- **Logical operator**
Combines Boolean values: &&, ||, and !.
- **Assignment operator or compound assignment**
The assignment operator is the equal sign =. A compound assignment combines = with an arithmetic operator.
- **Binary system**
The digit value number system of the computer using zeros and ones.
- **Built-in type**
A type that is available to you without #include.
- **Integer type**
One of short, int, long, or long long built-in types, each signed or unsigned.
- **Floating-point type**
One of the float, double, or long double built-in types; since C++23, [b]floatN_t.

- **Character type**
Most often `char`, but also with `wchar_t`, `char16_t`, and `char32_t` international characters; all built-in types.
- **Character string**
As literal `const char[]`, together with C often `const char*` in C++ `string`; in each case, also possibly with one of the other character types.
- **Truth value type**
The built-in type `bool` with its literals `true` and `false`.
- **Type inference with `auto`**
When defining a variable, let the compiler determine the type from the type of the expression of the initialization.
- **Overflow**
An attempt to change the value of a type beyond its value range; an overflow can occur in both the positive and negative range.
- **Literal**
A value specified directly in the source text.
- **`using` and `typedef`**
Defining a type abbreviation.

In this chapter, we will take a very quick “flight” over a simple C++ program. This will familiarize you with the most important elements and give you a better understanding of the things I will explain in the following chapters.

Let’s start with a simple C++ program.

```
// https://godbolt.org/z/Mdj7bGvar
#include <iostream>                // Include modules/libraries
int main()                        // main() is the start of the program
{
    int value = 100;               // Variable with initial value
    std::cout << "Divisors of " << value << " are:\n"; // Output of text
    for(int divider=1; divider <= value; divider = divider+1) // loop from 1 to 100
    {
        if(value % divider == 0) // test for conditional execution
            std::cout << divider << ", "; // only if the test is positive
    }
    std::cout << "\n";            // end of the loop
    return 0;                     // single output
}                                 // means end of program in main()
                                 // end of main()
```

Listing 4.1 A very simple C++ program.

If you compile and run this program, you will receive the following output on the screen:

Divisors of 100 are:

1, 2, 5, 10, 20, 25, 50, 100,

You can already see many basic and important things about C++ from this simple program.

4.1 A Quick Overview

I will briefly explain the individual program elements here so that you have the necessary basic knowledge with which we can then delve deeper.

4.1.1 Comments

As you can see, I have mixed program text and explanatory words here. The lines always start with program text, sometimes followed by a double slash `//`, and then come the explanatory words—the *comment*. In C++, you can write any text after `//`; the compiler ignores it (or, to be precise, interprets it similarly to a space) with a few exceptions. This allows you to tell other programmers, yourself, or posterity your intentions that led to the current program line.

4.1.2 The “include” Directive

The very first line of the example reads:

```
#include <iostream>
```

Use `#include` to inform the compiler that you want to use elements of a module in this file. The name between the brackets is the name of a header file containing the declarations of that module.

4.1.3 The Standard Library

I include the special module `iostream` because it contains `std::cout`. The program needs `cout` to generate the screen output. The `iostream` module is part of the *standard library* and is supplied with the compiler.

All names of the standard library begin with `std`, followed by the scope resolution operator `::`. This is an ugly, albeit precise term that nobody uses; the name *double colon* is fine too.

I will go into the possibilities of using the `std` identifier in more detail later. A large part of the book deals with the standard library itself.

4.1.4 The “main()” Function

Now let's look at the line that says `main`:

```
int main()
```

This defines a *function* with a special name. The `main` function is always the entry point into a C++ program: you can't do without it, and there can never be two. When your system executes the program, `main()` is called.

Otherwise, a function in C++ allows you to jump to this function from another location in the program and return to the call location later. Functions can take arguments—these are the *function parameters*—and return a result (see [Chapter 7](#)).

You can read the definition of this `main` function as follows:

- `main` should return a number—a value of type `int`, to be precise.
- The name of the function is `main`.
- The empty pair of round brackets in `main()` means that the function does not receive any parameters.
- This is followed by what the function actually does. This *function body* is always placed between two curly brackets `{ ... }`.

Functions other than `main` could return any other types.

The value returned by `main` is determined by the first `return` that the computer encounters when running the program. Here, this will always be `return 0`. The return value is therefore always 0.

Depending on the operating system, the return value can be evaluated. If your program is to be able to receive arguments on the command line, the round brackets for the parameters would not be empty. You will see later what you need to do for this.

4.1.5 Types

In C++, almost everything has a type, such as variables and intermediate results. The type determines which properties the construct has and which values it can accept.

In [Listing 4.1](#), only `int` is specifically mentioned as a type. The compiler will work out everything else itself. Two *variables* with this type are introduced in the program: `int value` and `int divisor`. Whenever you refer to these variables in the course of your program, you must take their type `int` into account. I will go into the exact properties of `int` later. For now, it is sufficient to know that `int` stands for an *integer*.

4.1.6 Variables

A *variable* is the name for a memory area that can hold a value. Yes, in C++ a variable must always have a type. When you use a variable for the first time, you must tell the

compiler its type. The type accompanies the variable as long as it lives and can no longer be changed.

In the program I use two variables: `value` represents the number whose divisor I output, and `divisor` is the number with which I check whether a division can be performed without remainder. First I define `value`. From this point on, I can use `value`, which is also done immediately for the output:

```
int value = 100; // Variable with initial value
std::cout << "Divider of " << value << " are:\n"; // Output of text
```

Because `value` is of type `int`, you can only use this variable for integers—that is, use it in calculations or change it.

The other variable is `divisor` in the `for` loop:

```
for(int divisor = 1; divisor <= value; divisor = divisor+1) // Loop from 1 to 100
```

The same applies to it as to `value`. Apart from the name, there are two other differences:

- `divisor` is actually changed in the program sequence. If `divisor = ...`, then it is assigned a new number.
- `divisor` is only known within `for`.

The *scope* of a variable is limited to its block, after which it is literally gone. And so gone that you could define a *new* variable `divisor` outside the `for`. This could then also have a completely different type. You can find an example in [Listing 8.4](#), where the variable `result` is redeclared (with the same type, though).

4.1.7 Initialization

In the example program, the equal sign `=` fulfills two purposes that are often mixed up. However, because the distinction is so important, I would like to make you aware of this at an early stage.

You can see the equals signs in the following example:

```
int value = 100;
int divisor = 1;
divisor = divisor+1;
```

The first two lines are each the *initialization* of a variable defined in the same breath. This point in time, at which you also define its type, is its *declaration*. You can only initialize something during the declaration.

The variable is already declared in the last line. You are therefore assigning a *new* value to an existing variable; this is why we speak of an *assignment*. With an assignment, you

cannot change the type of the variable. If the types are different, the compiler can perform a conversion within certain limits.

Assignment versus Initialization

The equals sign in the declaration is always an *initialization*.

It is only an *assignment* if the variable was declared somewhere else.

4.1.8 Output on the Console

With `#include <iostream>`, you've imported the part of the standard library that's responsible for input and output. The output to the console is done using the operator `<<`:

```
std::cout << divider << ", ";
```

On the left, you can see the variable originating from `<iostream>` with `std::cout`, which stands for the output on the console. To the right of each `<<` are the things you want to output. As you can see, you can concatenate `<<` like a normal plus `+` and thus output several things one after the other.

Some output is difficult to write in source code. C++ provides special mechanisms for this. One possibility is to *escape* certain characters with a *backslash* `\` (slash reversed). If you want to output a line feed, write `"\n"` in the source code, for example.

4.1.9 Statements

The curly brackets `{...}` of `main()` hold together a group of *statements* together, and they define a *statement block*. They form the boundary of what is executed for `main()`:

```
int main()
{
    ...
}
```

In between are *statements* that are executed one after the other. Statements are important basic elements in C++, and there are different types. Recognizing what a statement is and what type it is will quickly get you started with C++. In the next sections, you will get to know them all. At this point, I will show you which statements you can use. In [Listing 4.1](#), you will find the following:

- The *declaration* `int value = 100;` makes the variable `value` known and initializes it with an initial value—sometimes collectively called the *initialization statement*.
- With `cout << "divisor of " << value << " are:\n";`, it is an *expression* that outputs something to the console.

- This is followed by a *for loop*. It is used to execute other statements repeatedly. I will go into more detail about the `for` statement later, but note that the part that is to be repeated is again enclosed in *curly braces* `{...}` after the `for`.
- This is because the two braces that belong to the `for` are, with their content, a *compound statement* or a *statement block*. This again contains a series of statements that are held together by the enclosing braces. This grouping of statements has a special meaning in several respects. On the one hand, they can be repeated together in the `for` loop, and on the other hand, this grouping forms a *scope of validity* for the variables it contains.
- `if(value%divisor == 0)...` is an *if statement*, a *branching statement*. A condition is tested and the following *statement* `std::cout << divisor << ", ";` is only executed if this condition is true. As with the `for` loop in the example, we could have followed this with a statement block in `{...}` (that would have been good style). For demonstration purposes, the `if` is only followed by a single statement. This way I could save the surrounding `{...}` for a statement block.
- The *return statement* `return 0;` concludes this enumeration.

Statements are made up of *expressions*. An expression can have a value as a result at runtime. At compile time, the compiler knows the type of each expression. We will go into expressions in great detail later. Here are just a few examples from the program:

- `value%divisor` calculates the modulo—that is, the remainder of a division.
- `... == 0` checks the equality of two values.
- `divisor+1` is the result of an addition.
- `divisor = ...`, an assignment, is also an expression.

4.2 A Detailed Walkthrough

Having looked at an example program from start to finish in the previous section, I will now go into a little more detail. You will see the first formal definitions and learn to recognize the important language elements.

Let's build [Listing 4.1](#) a little and let's take a closer look at the elements we already know.

```
// https://godbolt.org/z/ceT6o5zxd
#include <iostream>    // for std::cin, std::cout, std::endl
#include <string>      // std::stoi

void calculate(int n) {                                // a separate function
    using namespace std;                               // for std::cout and std::endl
    /* output divisors */
    cout << "divisors of " << n << " are:\n";
    if(n == 0) { cout << "0\n"; return; }              // 0 is divisor of 0
```

```

    for(int divider=1; divider <= n; ++divider) { // for divider=divider+1
        if(n % divider == 0)
            cout << divider << ", ";
    }
    cout << endl;
}
int main(int argc, const char* argv[]) {           // Arguments for main
    /* Determine number */
    int value = 0;
    if(argc<=1) {
        std::cout << "Enter a number: ";
        std::cin >> value;                          // read into variable wert
        if(!std::cin) {                             // check whether reading worked
            return 1;                                // error on user input
        }
    } else {
        value = std::stoi(argv[1]);
    }
    calculate(value);                               // function call
    return 0;
}

```

Listing 4.2 This program asks its users for a number.

To begin with, you will see some new features: I have now introduced a separate `calculate` function. It receives a *parameter* of type `int`, which I address within the function under the name `n`.

Suddenly `main()` also has parameters—namely, `int argc` and `const char* argv[]`. This allows users to call the program on the command line with the number for which the calculation is to be performed. Here, `argc` contains the number of arguments that you can query with `argv[...]`. As `argv[0]` always contains the name of the called program, the first parameter is in `argv[1]`, and so on.

If the program is called without arguments, something must be entered at `std::cin >> value`.

`std::stoi(argv[1]);` converts a data type for the first time—in this case, a textual value (`const char*`) into a number (`int`). Then I use `calculate(value)` to call my own function `calculate` with the variable `value`.

4.2.1 Spaces, Identifiers, and Tokens

In addition to the comments, which are (as good as) meaningless for the compiler, there is also the space character, the tabulator, and the line break, which the compiler greatly simplifies when reading the source code: all these *whitespaces* are “collapsed”

and only considered as a single separator. It therefore does not matter whether you write `return 0;` or `return 0 ;`, insert line breaks, or even insert blank lines.

A small note on *line breaks*: this generic term covers the different variants that exist on the different operating systems. If you see a line break in the editor, different byte sequences end up in the text file on Windows and Linux, and on an old Mac it was another different one. While Linux stores the value 13 (CR, for *carriage return*), Windows stores the two values 13 and 10 (additional LF, for *line feed*). Most editors today can cope with both. But this difference is why you always have to specify whether you want to open a binary file or a text file when opening files programmatically.

Ultimately, it is only important that the compiler gets the smallest program units cleanly separated from each other—the *tokens*. If these are names (of variables, functions, etc.), then the boundary is clear—namely, where the name ends: the first character that is not suitable for a name is then the boundary. Names (or more precisely *identifiers*) contain letters and digits as well as the underscore `_`. But at the beginning, there must not be a digit. So `hello`, `Howdy`, `GoodDay`, `w3lc0me`, and `moin_moin` are all possible identifiers. On the other hand, `Lucky-Luke`, `3Investigators`, `Hardy Boys`, and `Tintin&Snowy` cannot be used as individual names. If you want to use umlauts as in `Bärenbrücke`, check whether your compiler can handle them; the standard has left some leeway here. If your program has to be translated by different compilers, it is better not to use umlauts or accents in the program text.

In addition to whitespace, a word can also be delimited by special characters such as braces or punctuation marks. Most of these characters are each a token, and you can insert any amount of space between these tokens without changing the meaning of the program. However, there are a few combination characters that only retain their meaning if they are written together—and thus count as a token. Pay particular attention to `>>` and `<<` as well as `++` and `--`, which were originally used to perform arithmetic operations, but which have been given additional meanings in C++.

When it comes to truth values, you will encounter `&&` and `||` as well as `==` and `!=`. In addition, `->` and `::` are also important for navigating in nested data structures. All other (special) characters are each a separate token. These include, for example, the following: `+ - * / = % , . ( ) [ ] { } < > : ; .`

The main reason that you should know where the tokens are in your source code is that you can omit and insert spaces without changing the meaning of the program. This is because you can insert any spaces, tabs, and line breaks between tokens.

Exceptions are the *literals*: with these, you write a fixed value directly into the source code. This can be an integer such as `100`, a floating-point number such as `99.95`, or a text such as `"Donald E. Knuth"`—and for all of these there are also variants in the notation. I will go into more detail in the discussion of the respective data types in [Section 4.4](#), where I discuss their literals in detail. Literals can sometimes look as if they consist of several tokens but count as a single one.

```
// https://godbolt.org/z/33cc3xfo6
#         include <iostream>           // # must be at the beginning of the line
int       main(
    ){
    std::cout
<<"This is "
        "text with <brackets>\n"      // string literal interrupted by new line
    ;
    /*type:*/ int
    /*Variable:*/ a_Value
    /*Init:*/ = 100;                  // inner comments
    std::cout<<a_Value<<"\n";}        // no spaces
```

Listing 4.3 A very unusually formatted piece of source code.

It becomes tricky when a construct is composed of several identifiers. For example, in `std::sin()` there is the *scope resolution operator* `::`. With templates, you will encounter pairs of angle brackets `<...>`—for example, in `numeric_limits<int>::max()` or `map<int, string>`. All elements are individual valid identifiers, but only together do they form a unit.

4.2.2 Comments

You have already seen comments with `//`. If there is not enough space to the end of the line, you can also have a comment extend over several lines by starting it with `/*` and ending it with `*/`. For example, like you see in the next example.

```
int main() {
    /* My first program. It was
    written by Max Muster.*/
    return /* The zero of success */ 0;
}
```

Listing 4.4 Comments with `/*` and `*/` can extend over several lines or interrupt a program line.

And because such a comment is delimited by `*/`, the program code can then continue on the same line, as you can see in the line with `return 0;`.

Within the comment, `//` may also occur. If you mainly use `//` for comments in your program, you can easily disable entire blocks of code in this way by enclosing the range with `/*` and `*/`—and reenabling it by removing it. The compiler would also swallow combinations such as `/* /* or // */`, but it is better to avoid all such embeddings; they confuse the reader and yourself.

In the examples in this book, I prefer to use `/*...*/` comments when I want to add text to the program code itself—without reference to this book. For example, in [Listing 4.2](#), I introduce program sections as follows:

```
/* Output divider */
...

and

/* Determine number */
...
```

4.2.3 Functions and Arguments

Don't be confused by the fact that the variable is named `value` when the function is called but is then called `n` inside the `calculate` function. As in mathematics, functions are defined in such a way that they provide their *parameters* with names under which you then use them—but what they are called with is a completely different matter. In mathematics, you can also have functions such as $f(x) = x^2 + 2$, $\sin(x)$, or compound interest.

$$\text{CompoundInterest}(\text{Dollar}, \text{Years}, \text{Interest}) = \text{Dollar} \times \left(1 + \frac{\text{Interest}}{100}\right)^{\text{Years}}$$

These functions all assign their own names to be used in the formula. You do not have to use these names when using them:

- $g(z) = z^3 - 2 \cdot f(z)$: For a nice curve, z becomes x in $f(x)$.
- $\sin(\pi)$: Here x is the number π .
- `CompoundInterest(1000, 12, 3)`: `Dollar` gets the value 1000 in the function, `Years` gets 12, and `Interest` 3.

Back to the C++ function from [Listing 4.2](#). As a reminder, it looks like the following example.

```
// https://godbolt.org/z/E4W9KhbnK
void calculate(int n) {                                     // a separate function
    using namespace std;                                    // for std::cout and std::endl
    /* output divisors */
    cout << " divisors of " << n << " are:\n";
    for(int divider=1; divider <= n; ++divider) {          // instead of divider=divider+1
        if(n % divider == 0)
            cout << divider << ", ";
    }
    cout << endl;
}
```

Listing 4.5 A custom C++ function.

In the function itself, compared to [Listing 4.1](#), from `/* Output divisors */` everything is almost identical. You will actually only find an alternative notation to `divider = divider + 1` in the increment part of the `for` loop with `++divider`. So far, you have used the *assignment*. Here, `++divider` is used as a *side effect operator* to change the value of `divider`.

4.2.4 Side Effect Operators

In [Listing 4.5](#), I used `++divider` to increase the variable by one.

This is the *prefix operator* `++`, which normally increases a variable by one (“increments”). This results in an *expression* whose value is the value of the variable increased by one. The result of the expression is unimportant in the update part of the `for` loop, but you can use the value within a more complex expression, as in the next example.

```
// https://godbolt.org/z/4Mr6fnaEr
#include <iostream> // cout
int main() {
    int basis = 2;
    int index = 10;
    int number = 3 * (basis + ++index) - 1; // index is incremented first
    std::cout << number << '\n';          // Output: 38
}
```

Listing 4.6 Prefix operators are executed before the calculation.

Here, `index` is first increased by one and the new value 11 is used in the further calculation.

If you change the value of a variable within an expression, this is a *side effect*. Never change the same variable twice within a statement with side effects!

```
int value = 0;
std::cout << ++value << ++value << ++value; // ⚡ no more than one ++value
```

The compiler will allow it, but the program may have different results on different systems. On a Mac I got “123” once here, on a Linux system “321”—and both are correct.

As you will see in [Chapter 9](#), you cannot always rely on the order in which expressions are evaluated. Therefore, you must not use such constructs. In C++, this minor inconvenience is accepted because it allows the compiler flexibility, which leads to better performance.

In [Table 4.1](#), I have listed the side effect operators.

I recommend that if you have the choice, you prefer the first two variants (prefix operators). This is because the last two variants (postfix operators) are generally more complex because the computer has to remember the old value in a temporary variable.

Operator	Description
<code>++var</code>	As seen, <code>var</code> increases by one and returns the new value.
<code>--var</code>	Decreases <code>var</code> by one and returns the new value.
<code>var++</code>	Increases <code>var</code> by one and returns the old value.
<code>var--</code>	Decreases <code>var</code> by one and returns the old value.

Table 4.1 Prefix and postfix side effect operators.

There is also the family of *side effect assignments*. All arithmetic operations are available in these variants; all are expressions and therefore have a value that you can use as part of a larger expression. However, they are not usually used as expressions, but as instructions like a simple assignment:

```
// https://godbolt.org/z/r5hjrqqK9
#include <iostream>
int main() {
    int var = 10;
    var += 2;
    var *= 3;
    var /= 4;
    var -= 5;
    std::cout << var << "\n"; // results in 4
}
```

In addition to `-`, `+`, `*`, and `/`, there are a few others that I will refer to in [Section 4.3](#) in more detail.

4.2.5 The “main” Function

Let me briefly simplify the smaller example program even further so that almost only `main` and `return` remain.

```
int main() {
    if(2 < 1) return 1;    // one return
    return 0;              // other return
}                          // end of main
```

Listing 4.7 A program that consists only of `main` and `return`.

Now you can see that `main` is also just a function—but a *special* function: every executable C++ program must have exactly one `main` function, because this is where the execution of the main part of what your program is supposed to do begins.

Without `main()`, the compiler and computer do not know where to enter the program. Only if you write a module or a library will you not have a `main()` function. If you then combine all the modules into a finished program, one of the modules—*exactly one*—will contain `main()`.

Put simply, the execution of your program begins with the first line of the `main()` function, directly after the opening curly brace `{`. However, there are all kinds of tasks that have already been done for you beforehand. The initialization of global variables is one of them.

If your program runs without errors, it runs until it encounters one of the `return` statements in `main()`. The number after the `return` is the *return value* of the function. Because `main()` is a special function that is ultimately called by the surrounding operating system, this value has a special meaning. On Unix systems, a 0 means that the program has run successfully and other programs can check this. And even on Windows, certain programs can signal success to other programs through the return value of `main()`. While the convention is that 0 means success, all values other than 0 mean failure. Beyond that, there are no general rules, except that the most commonly used value for failure is 1. There are also major differences in the possible value range between the systems, so you are on the safe side with values between 0 and 127.

A special feature of `main()` is that you can omit the 0 from `return 0`. You can even omit the whole line, in which case the program ends when the closing curly brace of `main()` is reached. Both are possible in `main()`—but only here and nowhere else.

Speaking of the special features of `main`, there is one more. As you can see in [Listing 4.2](#) and [Listing 4.7](#), you can declare `main` in a short and a long form:

- `int main()` or
- `int main(int argc, const char* argv[])`

If you are not interested in the call parameters of the executable program, opt for the first one. The second variant allows you to determine if users have invoked your program with arguments and to assess them accordingly.

Let's assume your program is called `divisors.exe`. In that case, (Windows) users can call up the program like this, for example:

```
$ divisors.exe 1001
```

Here `argc` will have the value 2, and with `argv[1]` you can access 1001.

There are only these two ways to define `main`. The form `main()` is a shortcut in case you are not interested in the call parameters of the program.

4.2.6 Statements

Let's take a closer look at what's between the `{ ... }` of our two functions: the *statements*. You already know that these are executed *one after the other*. This distinguishes them from *expressions*, for which the compiler can work its magic. Modern compilers and computers running your program do the craziest things to your program in the background, mainly to make it run faster. The only restriction on these transformations is that they must not change the meaning of the program. And this meaning is that the statements of your program are executed one after the other. This is exactly the difference from *expressions*, which are not executed in a certain order in this way. It is better to think that the expression takes on a value or that the computer *evaluates* it, but it does not *execute it* literally.

As a rule of thumb, a statement ends at its semicolon `;`. The most common exception to this rule is the *statement block*: the sequence of statements enclosed in curly braces.

In the next example, you can see an excerpt from the sample program as a reminder.

```
for(int divider=1; divider <= n; ++divider) // for loop
{
    if(n % divider == 0)
        std::cout << divider << ", ";
} // end of the loop block
```

Listing 4.8 Here you only set the outer curly braces.

To make it clear once again: the `for` loop is a *single* statement that goes from `for` to `}`. The same applies to the `if` branch. In the following listing, the condition part `if(argc==1)` and the branching part up to the last `}` belong integrally together and together form a *single statement*.

```
if(argc==1) { // start of the if statement
    std::cout << "Enter a number: ";
    std::cin >> number;
    if(!std::cin) {
        return 1;
    }
} else {
    value = std::stoi(argv[1]);
} // end of the if statement
```

Listing 4.9 This is what it looks like if you set all curly brackets.

Why is this important? Because this makes it clear where the loop can be used and how far it extends. Formally, a `for` loop always has the following format:

`for( loop-variable-definition ) statement`

And one of the formal possibilities for an `if` branch is this:

`if( condition ) statement`

As both of these are themselves statements, it is clear that I could have omitted some of the brackets in the `for` example. This is because statements can in turn contain statements. For example, the `for` statement block contains a single statement—namely, the `if` statement.

```
for(int divider=1; divider <= value; ++divider)
    if(value % divider == 0)
        std::cout << divider << ", ";
```

Listing 4.10 The “`if`” is a statement and does not actually require curly brackets.

This has the same meaning as the following:

```
for(int divider=1; divider <= value; ++divider) {
    if(n % divider == 0)
        std::cout << divider << ", ";
}
```

However, it is good style not to omit the curly braces for `if` and `for`. It would have been really good style and also equivalent to use the `{...}` of the statement block for `if` as well.

```
for(int divider=1; divider <= value; ++divider) {
    if(value % divider == 0) {
        std::cout << divider << ", ";
    }
}
```

Listing 4.11 It is better to put individual statements in curly brackets.

And a little more clearly: there are different ways of writing *where* the brackets are placed. Some prefer them as the last character in the line with the `if` or `for`; others put them on a separate line. As you have already read in the discussion of spaces, it does not matter whether you insert additional line breaks. It doesn't matter which of the options you choose, but it is certainly advantageous to follow one style consistently.

Although a semicolon belongs at the end of a statement, it should not be placed after the curly braces of a statement block. You have to be careful here, because unnecessary semicolons create an *empty statement*. This is harmless in most cases; for example, there are three empty statements between the semicolons in `;;;` without any effect. But a semicolon after an `if` means that the empty statement belongs to the `if` and ends it. In the following example, `cout << "smaller";` no longer belongs to the `if` and is now always executed:

```
if(value<10); // ⚡ a critical empty statement
    std::cout << "smaller";
```

It is therefore better not to enter any additional semicolons. Also make sure that you do not place a semicolon after the curly brackets of a statement block `{...}`. Even if it has no direct effect, it is better to avoid the following superfluous semicolon:

```
if(value<10) {
    std::cout << "smaller";
}; // ⚡ also avoid harmless empty statements
std::cout << "next";
```

Somewhat annoyingly, I will tell you in [Chapter 12](#) about `struct` and `class` that you *must* end their definitions with a semicolon. So by way of anticipation, you will be writing `struct type {...};`, and omitting the semicolon here would be a mistake. The reason is that this is a *definition* that always ends with a semicolon: it is not a *statement*.

4.2.7 Expressions

The statements are joined by the *expressions*. Expressions occur in quite a few places in C++ programs—not least as part of statements, but also in many other places that are not statements.

An expression is “a sequence of *operators* and *operands* that perform a calculation.” This is so dry that it can only come from the text of the C++ language standard. But what does it mean?

It means that something like `3+4` is an expression in which 3 and 4 are the operands and `+` is the operator. It can also get more complicated, such as in `(3+4)*PI/sin(x)`, where there are multiple operators and operands. Each operand must itself be a valid expression, and an expression is only valid if it is complete. Therefore, `1+2+3+` is *not* an expression, because the last `+` is missing an operand. Similarly, `3+4)` is not an expression because it is missing a parenthesis.

This example shows that expressions can be made up of smaller expressions. If you break `(3+4)*PI/sin(x)` apart, it contains the following expressions:

- `(3+4) * PI / sin(x)`
- `(3+4) * PI` and `sin(x)`
- `(3+4)`, `PI` on the one hand, `x` and even `sin` on the other
- `3+4`
- `3` and `4`

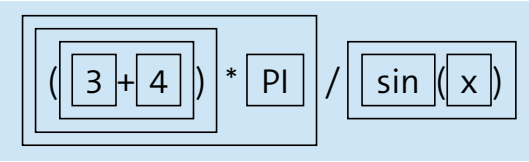


Figure 4.1 All subexpressions in one expression.

In [Listing 4.1](#) and [Listing 4.2](#), you will find all kinds of expressions, of which I can only list a few here:

- 100 is a *number literal*—a number that is written directly into the source code.
- "Divisors of " is a *text literal*—a text directly in the source code.
- `std::cout << "Divisors of "` is an expression that outputs to the screen (or writes to a file).
- `value % divider` calculates the remainder of the division `value / divider`.
- `value % divider == 0` checks whether this remainder was 0, and therefore `divider` is a divisor of `value`.
- 0 as part of `return 0;` is an expression that “calculates” the return of `main()`.

In a few places, we have something other than an expression, and I will briefly discuss this here. `int value = 100` is not an expression, but a *declaration* (`int value`) with an *initialization* (`= 100`). However, the initialization to the right of `=` must be an expression. `Return 0` is also not an expression, but a *statement*. It consists of the special word (*keyword*) `return` and, in this case, the expression 0.

One of the main characteristics of expressions is that each one has a well-defined *type*, and this also applies to each subexpression. Because types are very important in C++, understanding the types of expressions is also an important step toward success with C++.

More about Statements and Expressions

Because statements and expressions are such elementary language elements, we dedicate a later chapter to them ([Chapter 8](#)). Here you have been given an initial introduction and an overview.

4.2.8 Allocations

One of the most important expressions in C++ is the *assignment*. I did not mention assignment in the list of statements. That was not an oversight.

There are only a few assignments in [Listing 4.2](#) apart from the initializations—namely, the following example.

```
value = std::stoi(argv[1]);
```

Listing 4.12 Assigning the result of a function call.

And before I wrote `++divider` in the `for` loop, it said what you see in the following example.

```
divider = divider + 1
```

Listing 4.13 Assigning the result of a calculation.

In [Listing 4.12](#), `value` is assigned to the result of a function call. [Listing 4.13](#) contains a calculation whose result is assigned to the `divider` variable. The decisive factor is that the variable of the assignment already existed before and now receives a new value. This is done using the equals sign: the target of the assignment is on the left-hand side and an expression for the new value of the variable is on the right-hand side.

But an assignment falls into the category of *expressions*, and you can therefore use it wherever expressions are permitted. The fact that an assignment is used like a statement without being further embedded in a larger expression is actually a special case—but quite common in the case of assignments, as you can see in [Listing 4.12](#) and [Listing 4.13](#).

In the following example, the use of an assignment as an expression becomes clearer.

```
// https://godbolt.org/z/bh1MsPhje
#include <iostream>
int main() {
    int a = 3;
    int b = 7 + (a = 12) + 6;    // contains an assignment
    std::cout << a << std::endl;
}
```

Listing 4.14 An assignment is an expression with the type of the assigned variable.

Here, `a = 12` is the assignment. When executed, this expression has the value 12 and for the compiler the type of `a`—that is, `int`. The variable `b` is therefore initialized with the calculation `7 + 12 + 6`.

You should not take this as an example and use assignments as expressions everywhere. The assignment then becomes an expression with a *side effect* if, in addition to the main effect—changing `b`—another variable changes its value. We will later show you the few places where this is common in C++ and therefore there will be no surprises when reading and understanding.

Value versus Reference

In some programming languages, each variable is only a *reference* to an object. In Smalltalk, for example, this applies to all variables and in Java to everything that is derived from `Object`. There, an assignment does not change the object itself, but only assigns another reference. For example, if you have `string s = "a"; string t = "b";` in Java, then `s = t;` is an assignment. Now `s` and `t` “point” to the same `string` object.

This is different in C++: an assignment is always made to the *value* of a variable. If you have `string s = "a"; string t = "b";` in C++, then the assignment `s = t;` *copies* the *values* of the variables. Without further precautions, you always work with values in C++, not with references. Yes, you can also work with references in C++, but you must make this clear. As you’ll learn later, you then use `&` for references and `*` for pointers and addresses.

4.2.9 Types

In C++, almost everything has a *type*. Variables and constants have a type, as do parameters and also functions, data, classes, and expressions. Types are everywhere.

The type determines how much memory the computer must provide for a variable or the intermediate result of an expression, what kind of values it can hold, how these values are to be interpreted in memory, and what operations are permitted on them.

The latter is a strength of C++. This is because the main work of the compiler consists of processing the type information. First, it determines the type and finds out which things are permitted for this type. These can be function calls, operations, or conversions. What is meant by this program code? When the compiler finds this out, it applies its selection and thus determines the type for the expression.

This is why type and value always belong together: they complement each other. Conceptually, the type only exists at translation time, as a tool for the compiler. When you start the translated program (*runtime*), the types have disappeared and only the values exist.¹ So your program manipulates the values, and the compiler manipulates the types. Through its calculations, the compiler ensures that the desired operations on the values end up in the program.

You can determine the exact type of each (partial) expression. This depends on the operator and its operands. Sometimes it is easy to determine the type, sometimes not, but it is always possible. If it is difficult, you can get help from the computer. The right program for this is the C++ compiler! This is because it “calculates” the types of the expressions according to the rules specified by the C++ standard and your program.

¹ This is not entirely correct, as the compiler retains runtime type information (RTTI) in the program for certain language features. This can usually be switched off. However, it helps if you imagine it like this.

If you already know other programming languages, then you may be familiar with less stringent type concepts than those of C++. Some languages (such as Python, PHP, or JavaScript) ignore them almost completely and work well with them in their respective domains. Other languages (such as Java or C) have types but see their use somewhat differently. In C++, you are much more likely to use types to let the compiler help you write a correct program.

For example, if the program says `100+99` somewhere, then for the compiler this is an *operator* `+` with two *operands* `100` and `99`, each of type `int`. According to the rules that the compiler has built in, a `+` with two `int` operands again results in a value of type `int`, and therefore the overall expression also has the type `int`. At this point, the compiler does not calculate with the concrete numbers `100` and `99` and does not yet know anything about the meaning of `+`, but it only applies knowledge “about” operands and operators—their types.

If you had the right teacher in physics lessons at school, this may sound familiar to you. To check whether your result was probably correct when dealing with long formulas, you could only calculate with the units of measurement and quantities instead of the concrete numbers (“dimensional analysis”). If you came up with the wrong unit of measurement, you had made a mistake (in the calculation or when checking).²

For example:

$$\text{Force in } [\text{kg} \times \frac{\text{m}}{\text{s}^2}] = \frac{\text{Mass in } [\text{kg}] \times \text{Length in } [\text{m}]}{\text{Time in } [\text{s}^2]}$$

If you now want to know how much force is acting for a given mass of 8 kg, length of 12 m, and time of 4 s, then calculate $8 \cdot 12/4 = 24$ and $\text{kg} \times \text{m/s}$ at the same time and realize: “Oops, something's wrong with the seconds!” The correct calculation would have been $8 \cdot 12/4^2 = 6$, with the check $\text{kg} \times \text{m/s}^2$; that fits.

The compiler does something similar. You specify the rules of the calculation on types through the classes, functions, and templates of your program. These are supplemented by the standard library and other libraries that you use, as well as by the built-in rules. And just as dimensional analysis helped you in the physics exam, the compiler can help you to write a correct program by checking the type calculation rules, as in the next example.

```
// https://godbolt.org/z/9943MfExc
#include <vector>
class Image {
    std::vector<char> data_;
public:
```

² *Conversion of Units of Measurement*, Gordon S. Novak Jr, <http://www.cs.utexas.edu/users/novak/units95.html>, IEEE Trans. on Software Engineering, vol. 21, no. 8 (August 1995), pp. 651–661, [2014-01-31]

```

    void load(const char* filename); // loads image data
};
class Screen {
public:
    void show(Image& image);          // ✗ image should be const
};
void paint(Screen &screen, const Image& image) {
    screen.show(image);
}
int main() {
    Image image {};
    image.load("peter.png");
    Screen screen {};
    paint(screen, image);
}

```

Listing 4.15 The compiler helps to write correct programs by checking the types.

The omission of “to the power of 2” in C++ corresponds perhaps to a missing `const`: The `Screen` class has a method `show` with which it paints an image. You would think that the image `image` would not be changed when painting. It is therefore likely that the type of the `show` parameter should have been `const Image&` rather than `Image&`.

A forgotten `const` alone is not yet critical, mind you. But what would happen if someone “accidentally” placed the call `image.load("paul.png");` in the `show` method? That would change the image. This cannot happen with the `const` parameter: the compiler reports an error.

You should use *type safety* wherever you can. It represents one of the most powerful advantages of C++ over good old C. There, the transformations possible on types were (and are) much more restricted—and the checks much more succinct.

First Simple Types

Without perhaps realizing it, you have already dealt with all kinds of types in the small examples we have discussed. We will now cover the most important ones.

The “`int`” Type

In this chapter, `int` is a representative of all the types that you can use in C++ to represent *integers*—that is, negative integers, zero, and positive integers. However, a computer has limitations when it comes to its number range. With `int`, you have a different number of *bits* available depending on the system. If you are working on a current system, it is probably 32 bits, but it could also be 16, 64, or a completely different number. With 32 bits, `int` can store integers from about -2 billion to +2 billion. You can find out

exactly with [Listing 23.29](#). If you want to cover other number ranges, you should take a closer look at the relatives `char` and `short` as well as `long` and `long long` in [Section 4.4](#).

The “bool” Type

While an `int` variable can assume many states, a `bool` variable is fixed to one of two states: `true` or `false`.

Comparisons are expressions whose result is of type `bool`, and you can store their result in a variable of this type instead of using it directly.

```
// https://godbolt.org/z/zK9Ke9n48
#include <iostream>                                // cout
int main(int argc, const char* argv[]) {
    bool withParameters = argc > 1;                // comparison result stored
    if(withParameters) {                            // ... and used
        std::cout << "You have called the program with parameters.\n";
    }
    return 0;
}
```

Listing 4.16 Variables of type `bool` can store the result of a comparison.

The Types “const char*” and “std::string”

These two very different types have the same purpose: their variables represent strings—that is, text strings, like texts, messages, notes, names, and so on. Depending on the task, sometimes the “pointer variant” `const char*`, which cannot deny its origin in C, is suitable, and sometimes the `std::string` “class” from the C++ standard library is suitable.

Pointers, Addresses, and Memory Areas

Note the asterisk `*` in `char*` or `const char*`. This tells you: “I am only storing the address of something.” This means that there is a `char` where the variable points to. There is also `int*` as a pointer to an `int` and so on.

When it comes to character *strings*, however, another piece of information is missing because the pointer only contains the address of the *first* character; others may follow. In C, it is common for a text to extend to the next `char` with the value 0, also written `'\0'`; this is then a C string. The C function `printf(const char* fmt, ...)` works in this way, for example.

Other functions prefer to take pointers and a length, such as `memcpy(void* dest, const void* src, size_t n)`. Here, starting from `src`, the next `n` bytes are copied to `dest`.

The third variant, using pointers to define a range, is often used in C++ for containers and algorithms. For example, you can use `std::find(const char* beg, const char* end, char c)` to find the character `c` between `beg` and `end`.

If you only want to output a fixed character string, you usually work with `const char*` because in this form many program parts exchange character strings with each other. You have already seen the parameter for `main`, which I described in [Listing 4.16](#) for the example `argv`. There, the system may bring several program arguments into the program in the form of `const char*` strings. And even if you write a string with `"..."` directly into the program text, the compiler stores this “string literal” as a `const char*` in the computer's memory. (More precisely, it is a `const char[]`, but this is very similar and should suffice at this point.) As long as you only want to output this text, as in

```
std::cout << "You have called the program with parameters.\n";
```

then you do not need to worry about this.

For most cases beyond that, I recommend that you use `std::string` instead. If you want to save, copy, or manipulate strings, this is the best choice. Especially with regard to memory management—the dynamic creation of new strings—it is much easier to use the C++ class `std::string`.

For the sake of simplicity, a `const char*` literal is also used when initializing `string`, but since C++14 you can also append an `s` to a `"..."` literal and thus turn it into a real `string` literal. To do this, however, the block must contain `using namespace std::literals` or one of the alternatives somewhere, as in the next example.

```
// https://godbolt.org/z/chTTqPjrM
#include <string>
#include <iostream>
int main() {
    std::cout << "string_literals\n";
    // using std::literals::string_literals::operator""s;
    // using namespace std::string_literals;
    // using namespace std::literals::string_literals;
    using namespace std::literals;
    std::cout << "Real string\n "s;
}
```

Listing 4.17 This is how you can write a C++ string as a literal in the source code.

This rarely makes a big difference but can sometimes be important.

Because you will be using `std::string` a lot (at least in this book), allow me to abbreviate it to `string`.

Parameterized Types

Some types contain angle brackets `<...>` in their name. You have seen

```
std::vector<char>
```

in the example. Even if this looks different from `int`, `bool`, or `string`, the *whole construct* is still a single type. The part in the brackets is a parameter to form the overall type. Here `vector` is the surrounding or main type, and `char` is the parameter. Together, the two are a “vector of `char`.”

You could also form a `vector<int>` and thus obtain a different type. This is important, for example, if the compiler is to decide which functions are available for selection or what the result of an operation is. As I have already mentioned, the compiler calculates with types in the translation phase; the program only calculates on the values at run-time.

One thing is particularly important here: with parameterized types, what is inside the angle brackets must always be known at compile time. This is only logical as the compiler calculates with types at compile time and the entire construct, consisting of the main type, brackets, and parameters, makes up the type. If you remember this, you will not be surprised when experimenting. For example, there is also the parameterized type `std::array<...>`, which receives a number as the second template parameter. The number must be a constant or something that the compiler can already calculate, such as `3+4`. For example, you can write `std::array<int,5>` but not `int n=5; std::array<int,n>`, because `n` is a variable and as such cannot be part of the type.

Note

A type must be fixed at compile time.

Class Template Argument Deduction

You do not always have to specify the arguments of a parameterized type. If you call a constructor, the compiler can infer the types themselves, as in the next example.

```
// https://godbolt.org/z/GbbcnYbqv
std::vector vec { 1, 2, 3 };           // instead of vector<int>
std::tuple tpl { 5, 'x' };           // instead of tuple<int,char>
std::shared_ptr<int> ptr { new int(5) };
std::shared_ptr ptr2 { ptr };         // instead of shared_ptr<int>
```

Listing 4.18 Since C++17, the compiler determines the type parameters of class templates based on the constructor arguments.

This allows you to keep your code concise and clear in some cases. However, this is not always possible; you will be naming the type parameters often enough yourself.

The class template argument deduction (CTAD) feature has been available since C++17. You can read more about this in Listing 23.33. If you are using an older compiler, write out the parameters of the types.

4.2.10 Variables: Declaration, Definition, and Initialization

It makes a difference whether you write

- `int value;`, a *declaration*;
- `value = 0;`, one *assignment*; or
- `int value = 0;` a *definition* with *initialization*.

There are also variables that you can no longer change once they have been introduced. They are constant—and are then actually called *constants*. Apart from this fact, constants are *also* variables for the compiler—that is, a subset of them in terms of “set theory.” You can never reassign anything to them, so you only have the option of initializing them with a value when you declare them. I will go into more detail about `const` later, but you can probably guess the main purpose of this addition.

```
int main() {
    const int fest = 33; // Initialization as constant
    fest = 80;          // ✗ an assignment is impossible
}
```

Listing 4.19 You cannot assign anything to some variables; you can only initialize them.

There is another way in which you can distinguish between assignment and initialization or make them distinguishable yourself. With C++11, *unified initialization* was introduced. Wherever you initialize variables, you can now use curly brackets instead of `=`.

```
int index = 1;           // old style, looks like an assignment
int zaehler { 1 };      // C++11 style, clearly an initialization
int counter = { 1 };    // in the C++11 style, the "=" is optional and is ignored
```

Listing 4.20 Instead of `=` you can use `{...}` for initialization.

There was previously a risk of confusion not only with the assignment, but also with a handful of other C++ constructs.

Risk of confusion()

This is a preview for things to come, but I'll give you an example. You can pass a temporary `Thing` to a function `f` that you initialize with the value 12—that is, call its constructor:

```
f(Thing(12));
```

If it is not intended to be temporary, for example, this works:

```
Thing d = Thing(12);
```


For a constructor without arguments, this can be done in one of these ways:

```
f(Thing());
Thing d = Thing();
```

The following is also fine:

```
Thing d(12);
```

You now have `d` that has been initialized with 12. Now you could conclude that you initialize such a `d` without constructor arguments like this:

```
Thing d(); // ✗ Error: Declaration of d as a function
```

And you would be wrong! You have now declared a function `d` that takes no arguments and returns a `Thing`. Please note: You have *declared* it and not *defined* it. This is similar to `int mydata();` as a function declaration in a header file. As soon as you want to use `d` in any way, such as with `d.print()`, the compiler complains with the most obscure error messages, such as that “`d` has not yet been defined” or “a function call was expected.”

If you want to try it out, you don’t need a `Thing`. Try it with `int` or short:

```
// https://godbolt.org/z/3G5Po9zEj
int main() {
    int d1(12); // defines an int and initializes it with 12
    int b1 = d1+1;
    int d2(); // ✗ declares a function d2 without parameters, return int
    int b2 = d2+1;
    int d3{}; // defines an int and initializes it with 0
    int b3 = d3+1;
}
```

My error message today:

arithmetic on a pointer to the function type 'int ()'

The solution is to consistently utilize `{...}`, as `Thing d{};` does what you expect.

Old Equal Sign or Standardized Initialization?

The new initialization with `{...}` is supported by all current compilers. So you have a choice. The “old style” cannot be eradicated (and it is debatable whether this would make sense), but you should use the new style wherever you are not just initializing a native data type such as `int`. This will equip you for the tricky cases.

4.2.11 Initialize with “auto”

A word about initialization. It always consists of

- the type of variable you are initializing,
- the name of the variable, and
- an expression for the first value of the variable.

The type of the expression is often the same as the type of the variable. Sometimes you let the compiler perform an *implicit conversion*. However, you can often save yourself the work of specifying the type of the expression again. You can use `auto` instead. The compiler then takes the type of the expression to determine the variable type. This is called *type inference*:

```
auto number = 10;
auto value = 3.1415 * sin(alpha);
for(auto it = container.begin(); ... ) ...
```

The variable `number` here becomes an `int`, while `value` becomes a `double`, and it is of the type that `begin()` returns, probably an iterator of the variable `container`.

Since C++20, however, you can further restrict the type with a *concept*. This allows you to specify that the type of variable must fulfill certain criteria. Many concepts are available in the standard library for this purpose, such as `integral` and `floating_point`:

```
auto integral number = get_number();
auto floating_point pointy_number = get_ampere();
```

If there was only `auto number` here, then `number` could also be a `double`—which would perhaps lead to problems with `number` in the further course of the calculation. On the other hand, if you were to completely define the type with `int number`, you would lose the option of extending the return value of `get_number()` to `long`. In addition, the line of code in question could also be in generic code, where the return type of `get_number()` can be `int` or `long`, but you want to prevent `get_number()` from returning a `double`.

The advantages of a concept together with `auto` are documentation and security against surprises.

In addition to the built-in concepts in the standard library, you can also define your own concepts.

Whether with a concept or without—`auto` defines the type of the variable as if you had explicitly written the type. It is by no means the case that `auto` is its own mutable type. Once set, the variable retains its type.

The compiler determines the type `int` for `val`. A later assignment of a text literal is an error:

```
auto val = 12;
val = "Name"; // ✗ Error: Assignment of "const char*" to "int"
```

You can find out exactly how `auto` works in [Chapter 12, Section 12.12](#).

4.2.12 Details on the Include Directive

Includes always begin with a hash sign #. Such a character at the beginning of a line stands for an instruction to the *preprocessor*. If you remember [Chapter 2, Figure 2.1](#), you are already familiar with this phase of the translation.

The file that is included here with `#include` is inserted into your program by the preprocessor almost as if by copy and paste. The effect is that all declarations from that file are now part of your program.

There are other preprocessor directives, but `#include` is the most important one for you. The others can be found in [Chapter 21](#).

Includes always belong at the top of your source files. In [Listing 4.2](#), I have used two of them.

```
#include <iostream> // cin, cout
#include <string>    // stoi
```

Both `<iostream>` and `<string>` are part of the *C++ standard library*. They therefore contain things that are not part of the *core language*. Something like `int` is always present on all systems without the need for an include. The standard library will usually be present—and we assume that it will be in this book—but not always.

As you introduce many identifiers into the current source code with `#include`, it is helpful to add a comment to indicate which of them you are using in this translation unit. This allows readers to find out where an unknown identifier comes from by simply searching the file. Some module names are self-explanatory. For example, `#include <vector>` imports `vector`; there is no need to explain this.

You can recognize the includes of the standard library by the fact that the name does not have an extension such as `.h` or similar. The convention is that only *header files* of the standard library without an extension are used.³ If you create your own, then append an `.h`, `.hpp`, `.H`, `.hh`, or `.hxx`. In practice, `.h` is the most common, even if some programmers do not like this, as the header does not differ from C headers from the outside. The `.hpp` file extension is often chosen as an alternative. Regardless of which extension you use, you name it in the `#include` directive:

```
#include <iostream>           // Module of the standard library
#include <asteroids.h>        // Module of a third-party provider
#include "myModule.hpp"      // Module of the current project
#include "algo/myModule.h"   // in a subdirectory
```

Write the name of the header file in angle brackets if the file is *installed* on your system; the compiler should search for it. Use the quotation marks `"..."` if the header file is part

3 However, you will find well-known libraries that also do not use an extension, such as the Qt library.

of your current project and not installed somewhere. Some compilers then search for the file in the current project first.

Don'ts for Includes

Pay attention to upper- and lowercase when naming the file, as this is observed on some systems, dear Windows users! If the header file is located in a subdirectory, always use / to separate it:

```
#include "MY_MODULE.H"           // ✗ not good if the file is called "myModule.h"
#include "algo/myModule.h"       // ✗ also use "/" under Windows
#include "c:/project/myModule.h" // ✗ no absolute paths
#include "../algo/myModule.h"    // ✗ no relative paths with ".."
```

Instead of using the absolute path or .., you should store the header files somewhere else in your project or/and must adjust the search paths for header files in the compiler options. This will make it easier for you to move your project and reorganize it if necessary.

One more note on brackets versus quotation marks: Originally, `<...>` versus `"..."` meant that the header may be part of the compiler and not a file at all. However, it has become accepted that the brackets are also used if it is simply a header file that you are expected to have installed on the system. Therefore, only use `"..."` for files that are part of your project and should have priority over other libraries. Whether the priority works depends on the compiler used.

4.2.13 Modules

Modules were introduced with C++20. Instead of “copying” textual `#include` directives into a translation unit, divide your code into interface and implementation areas. To do this, use the new keywords `module` and `import`. You can find the exact usage in the C++ reference and in tutorials such as <https://youtu.be/tjSuKOz5HK4> by Dos Reis. However, there is still no “best practice,” and the standard library is not yet divided into modules. With C++23, the two standard `std` and `std.compat` modules were introduced, which contain *everything* from the standard library. The `std` module contains the symbols that can be accessed with `std::`, such as `std::vector`. The `std.compat` module contains the identifiers of the global namespace, such as `::fclose`.

But today, in 2024, not much of this has been implemented with the major g++ 13, clang++ 18, and msvc++ 19.37 compilers, and some implementations are incompatible with others. As long as it is not foreseeable that modules are within reach for you, I refer you to the highly topical media if you are interested.

However, I would like to briefly pass on some recommendations from Nico Josuttis on how you can best work with the emerging conventions and have as few problems as possible when porting to another platform:⁴

- *Interface files* should have the `.cppm` extension (do not use `.ixx`; use `/interface` for MSVC).⁵
- Keep *module implementation files* as `.cpp`.
- *Internal partition implementation files* should have the `.cppp` or `.cppm` extension.

But if you use them, modules and includes are not mutually exclusive. As long as you adhere to certain schemas, you can use both in parallel—and will probably have to for a long time to come, because even if modules are the better alternative, includes will still be around for a very long time for historical reasons.

4.2.14 Input and Output

Now you know all the important elements of the program, except—some would say—the most important ones. After all, what is a program without an *input* and *output*?

You have already seen that `<<` is used as an operator for the output. And `#include <iostream>` was also included because of `std::cout`. Here is an excerpt from [Listing 4.2](#), which is mainly about the output:

```
std::cout << "Divisors of " << n << " are:\n";
for(int divider=1; divider <= n; ++divider) {
    if(n % divider == 0)
        std::cout << divider << ", ";
}
std::cout << std::endl;
```

You can see that you can simply string several `<<` together. More specifically, `std::cout << "divisors of "` is an expression that does the output as a side effect and then returns `std::cout` again. This effectively executes `std::cout << n`, again with the output and the result `std::cout`. Last but not least, the third operator of the statement comes into play and `std::cout << " are:\n"` is executed. What is special about this operator is that it contains a *newline*. In C++ strings, this is represented by `\n`. Exactly this newline was omitted within the loop in the output; therefore, the numbers appear on one line.

With `std::cout << std::endl`, the program outputs the line break at the end of the list. But, oops, where is the `\n`? This time there is a special element of `std::cout` in the listing, the *manipulator* `std::endl`. This ensures the line change and also guarantees that the system does not buffer the output. It forces everything you have output so far to actually appear on the screen. This is because (screen) output is time-consuming, and

⁴ C++20: The Complete Guide, Nico Josuttis, <https://leanpub.com/cpp20>, Leanpub, 2022

⁵ Cppmodules, Nico Josuttis, <https://github.com/josuttis/cppmodules>, November 30, 2023

the system tries to summarize as many steps as possible and complete them all at once. With `std::endl`, you ensure that the system really outputs everything that has occurred.

4.2.15 The “std” Namespace

Now there is one more thing I need to explain to you before we dive deeper into the individual topics. In [Listing 4.2](#), you can see `using namespace std`. With this tool, you save yourself the `std::` in front of identifiers that you would otherwise have needed.

```
#include <iostream>                // for std::cin, std::cout, std::endl
#include <string>                  // for std::stoi
void calculate(int n) {
    using namespace std;          // for std::cout and std::endl
    /* output divisors */
    cout << "divisors of " << n << " are:\n"; // cout instead of std::cout
    for(int divider=1; divider <= n; ++divider) {
        if(n % divider == 0)
            cout << divider << ", ";        // cout instead of std::cout
    }
    cout << endl;
}
```

Listing 4.21 You can include a namespace to make program text shorter.

Normally, the compiler checks for each identifier whether its name exists locally or on the outer levels. For example, `divider` is a local variable and `n` is a parameter within the function. If you were to use `cout` without `using`, the compiler would check whether there is a local variable or a parameter with this name, then whether a global variable `cout` exists, and then inform you with an error that this is not the case—and that the identifier is therefore unknown.

With `using`, for each identifier that could not be found, the system tries whether it works with a preceding `std::`. This can save you a lot of typing, where you would otherwise type a lot of `std::`s.

The `using namespace` directive has an effect within the range in which it is defined—in the example, only within the `calculate` function. You can also write `using namespace` globally, in which case the scope is larger.

```
#include <iostream>
#include <string>
using namespace std;           // ⚡ has a global effect; works, but is critical
void calculate(int n) {
    /* output divisors */
    cout << "divisors of " << n << " are:\n";
```

```
// ...
}
// ... also in other functions ...
```

Listing 4.22 You can also use “using namespace” globally, but you should rarely do so.

Does that sound so tempting that you want to use `using namespace ...` all the time? Don't do that! With `using namespace ...`, you no longer know where the identifiers of your program come from. For example, if you are not yet familiar with the standard library and come across a free-standing `cout`, how are you supposed to know that it is actually `std::cout`?

Avoid Global “using namespace”

Do not use `using namespace ...` at a global level in a file. When reading each identifier, you ask yourself where it comes from, and you have no chance of finding out even by searching in the current file.

Within a block—for example, locally in a function, as I have shown in [Listing 4.21](#)—a `using namespace ...` is not frowned upon, however. The range for wondering which namespace an identifier comes from is limited enough. And the probability that several using namespaces are active at the same time is lower.

But there is a solution: do not get all the identifiers of a namespace, but only the identifiers that you need, as in the following example.

```
#include <iostream>                // cin, cout, endl
using std::endl;                  // applies globally in this file
void calculate(int n) {
    using std::cout;              // applies locally in this function
    /* output divider */
    cout << "divisors of " << n << " are:\n";
    for(int divider=1; divider <= n; ++divider) {
        if(n % divider == 0)
            cout << divider << ", ";
    }
    cout << endl;
}
```

Listing 4.23 Get individual identifiers with “using.”

Including identifiers in this way causes few problems. If a standalone `cout` appears somewhere in the file, you can find the original namespace with a simple search within the file.

In This Book

As the listings in this book rarely become long and confusing, I will make an exception here. It saves space and repetition to write using `...` at the beginning after the includes. Therefore, I will mainly abbreviate `std::cout`, `std::endl`, and `std::string` but will write out other identifiers with `std::` so that you know where they come from.

4.3 Operators

One- and two-character operators are very often used in expressions. `3+4` is a simple example, but an expression such as `!isBad && (x >= x0) && (x <= x1)` could also appear. You can achieve a lot in C++ with such sequences of operators and operands. Now that you know a lot about variables, types, and expressions, you should get to know the possible operators.

As an example, I will explain operators mainly using the `int` and `bool` types so that you get to know the repertoire first. However, many operators can also be applied to other types. These include built-in types, such as `float`, but also types from the standard library, such as `std::string` and `std::stream`.

In C++, you can define your own types that also support operators. It is up to you to ensure that these then also do something that applies to the built-in types. For example, we expect `+` to perform an addition—like with an `int`. However, the class `string` uses `+` for concatenation, which is still “addition-like.” But you can, if you want, write a class `Image` that stores itself in a file with `+` followed by a `string`. (Please don't do this!) We will deal with this in a suitable place. Here you will first find out what operators there are.

Operators for Built-In Types Cannot Be Overwritten

When I describe operators in this chapter, I therefore mean those for the built-in types. You cannot change these in C++. There is already a `+` for `int`, and you as a programmer cannot change its meaning. In some places, I refer to the types of the standard library, but for many things, the reference must serve.

4.3.1 Operators and Operands

An *operator* is something that behaves like a function, but without being one. What is different is that you would expect a function to have its name at the front and the arguments in brackets after it, i.e. `func(arg1, arg2)`. With a little creative freedom, you could write `func` in the middle and omit the brackets, and you would get `arg1 func arg2`. And

if the function name were `+` instead of `func`, you would have an operator `+` with its two operands `arg1` and `arg2`.

Most operators are written with symbols, such as `+` or `<<`. Many have two arguments and are therefore called *binary operators*. In C++ it always looks like this:

- *Operand operator symbol operand*

If they are *unary operators*, the operator is placed before the operand as in `-4` or, more rarely, after it as in `idx++`:

- *Operator symbol operand*
- *Operand operator symbol*

There are other forms of operators, which are explained next.

4.3.2 Overview of Operators

The operator symbols can be divided into several groups:

- **Arithmetic operators**

These are the four basic arithmetic operators `+`, `-`, `*`, and `/`, plus `%` for the remainder of the division (modulo). You can influence the sign with the unary operators `+` and `-`.

- **Bitwise arithmetic**

You can combine numbers bit by bit with `|`, `&`, `^`, `~`, `<<`, and `>>`.

- **Compound assignment**

In addition to `=`, there are also assignments that perform another operation at the same time. These are *compound assignments*: `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, and `|=`.

- **Increment and decrement**

The two single-character operators `++` and `--` are each available in a prefixed and a postfix variant (prefix and postfix). If possible, prefer the prefix variant, as it does not require a temporary variable.

- **Relational operators**

Relational operators perform a comparison and return a truth value `bool`: `==`, `<`, `>`, `<=`, `>=`, or `!=`. If you are working with the standard library, `==` and `<` are the most important, as many algorithms only use these two to derive the others if necessary. This is important if you want to make your own data types fit with the standard library.

- **Logical operators**

`&&`, `||`, and `!` combine truth values to form more complex expressions.

- **Pointer operator and dereference operator**

You use the unary operators `&` and `*` and the binary operators `->` and `.` to address and to dereference. This means that you retrieve an address, turn an address into data, or access a structure. You will see their use in detail later.

■ Special operators

You can use `?` and `:` together to write an either-or expression. The comma `,` can be used as a sequence operator in expressions. With C++20, the spaceship operator `<=>` is added, which helps to implement the relational operators.

■ Function-like operators

Strictly speaking, some special types also belong to the operators that have real names and are used like functions. These are type conversions such as `(int)value`, `sizeof(a)`, and some others. The fact that these are operators means, for example, that you cannot request the address from `sizeof`. You can only give a function values and variables as parameters, but with `sizeof()` you can also use a type, such as `sizeof(int)`. Strictly speaking, the brackets are also optional; you could write `sizeof a`, for example, but that would be very unusual.

4.3.3 Arithmetic Operators

You can calculate with numbers as normal in C++. In addition to the basic arithmetic operations `+`, `-`, `*`, and `/`, there is also `%` for the division remainder. If you do not use brackets, then *dot before dash calculation* applies, as you can see in the next example:

```
// https://godbolt.org/z/cEGGfoeT7
#include <iostream>
int main() {
    std::cout << "3+4*5+6=" << 3+4*5+6 << "\n";           // dot before dash; = 29
    std::cout << "(3+4)*(5+6)=" << (3+4)*(5+6) << "\n";     // Parentheses; = 77
    std::cout << "22/7=" << 22/7
        << " remainder " << 22%7 << "\n";                 // 22/7 = 3 remainder 1
    for(int n=0; n < 10; ++n) {
        std::cout << -2*n*n + 13*n - 4 << " ";             // with unary minus
    }
    std::cout << "\n";
    // output: -4 7 14 17 16 11 2 -11 -28 -49
}
```

Listing 4.24 Arithmetic operators in the application.

What was shown here for `int` works with all integer types. And except for `%`, it also works with all floating-point types (`float` and the like). In the standard library, you will find `std::complex<>`, with which you can also use these operators.

Many types use the plus operator `+` for joining. For example, you can use

```
std::string before="Robert";
std::string after="Reynolds";
```

with `before+" "+after` to make a new string, "Robert Reynolds".

4.3.4 Bit-by-Bit Arithmetic

Bit-by-bit arithmetic probably looks a bit strange at first:

```
int a = 41;      // decimal 41
int b = a & 15;  // results in 9
```

The explanation is that numbers are represented in the computer as a sequence of 0s and 1s—that is, as bits. The decimal 41 is 101001 in bit representation—that is, in binary.

Binary System

Because 10 is the base in the decimal system, we write “four hundred and twelve” as 412 as an abbreviation for $4 \cdot 10^2 + 1 \cdot 10^1 + 2 \cdot 10^0 = {}_{10}412$. For a computer, with base 2, this is $1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$, abbreviated to ${}_210011100$. The subscript 10 or 2 indicates the number system in which the number is represented.

Any conversion from the decimal system to another is simply done by repeatedly dividing by the base and noting the remainder until the value reaches zero. For example, to convert 10412 to the binary system, divide repeatedly by 2:

```
412 / 2 = 206, remainder 0
206 / 2 = 103, remainder 0
103 / 2 = 51, remainder 1
 51 / 2 = 25, remainder 1
 25 / 2 = 12, remainder 1
 12 / 2 = 6, remainder 0
 6 / 2 = 3, remainder 0
 3 / 2 = 1, remainder 1
 1 / 2 = 0, remainder 1
```

Read the remainders from bottom to top and write them down as a binary sequence. The lowest 1 therefore represents the highest digit: ${}_2110011100$.

Of course, this is also possible in C++. [Listing 4.25](#) is partly a preview of things you will get to know, but I think it is still instructive.

```
// https://godbolt.org/z/nMcsbxvvv
#include <iostream>
void printBin(int x) {
    while(x>0) {      // done?
        int a = x/2;  // division by 2
        int b = x%2;  // modulo, remainder of the division
        std::cout << x << " / 2 = " << a << ", remainder " << b << '\n'; // Output
        x = a;
    }
}
```

```
int main() {
    printBin(412);
}
```

Listing 4.25 Programming example for the conversion of an integer into a bit sequence.

However, only a few people can do this in their heads. I myself calculate smaller numbers in my head the other way round: starting from the top, I subtract the powers of 2—32, 16, 8, 4, 2, and 1—until I reach zero. In $_{10}25$, 16 fits, then 8, then 1, which results in binary $_211001$.

The computer often calculates with 32 bits for an `int`. It therefore fills the front with 0s—that is, $_200000000\ 00000000\ 00000001\ 10011100$ for $_{10}412$.

Negative Integers and the Two's Complement

For signed integers, such as `int`, which is implicitly `signed int`, the most significant bit represents the sign; 0 stands for a positive number and 1 for a negative number.

If the sign is positive, the remaining bits contain the value of the number as usual. However, if it is negative, the interpretation of the value bits changes. The computer stores this in the *two's complement*. To convert the bit representation of a negative binary number into a decimal number, proceed as follows:

- Invert all value bits.
- Convert this to a decimal number as usual.
- Add one.
- Set the negative sign.

The value bits of $_21100$ are $_2100$, inverted therefore as $_2011$. This corresponds to $_{10}3$, which after the addition of one gives the value $_{10}4$. In total, $_21100$ therefore represents $_{10}-4$ in two's complement.

The two's complement makes optimum use of the value range of the data types and has proven to be the most practical for computer calculations. Incidentally, a side effect of this representation is that the smallest negative number that can be represented always has an amount that is one greater than the largest positive number that can be represented. The range of 16 possible values for four bits goes from -8 to 7. For eight bits, the range goes from -128 to 127, and so on.

The arithmetic is now the bit-by-bit combination of the numbers written in the two systems. If you think of the 1 as “true” and the 0 as “false,” then you can perform the bitwise operations for *AND*, *OR*, and *XOR* yourself. For the sake of simplicity, I will limit myself in the example in [Table 4.2](#) to 4 bits for unsigned integers.

Operation	Binary	Decimal
a	1001	9
b	0011	3
And a & b	0001	1
Or a b	1011	11
Xor a ^ c	1010	10
Not ~b	1100	12

Table 4.2 Bitwise arithmetic with a (hypothetical) four-bit-wide unsigned int.

The << and >> operators should no longer scare you: simply shift the bit representation of the first operand to the left or right by the number of bits of the second operand. In the computer, it is the same as if you were multiplying or dividing by two just as often:

- `345 << 3` is like `345*2*2*2` and results in 2760.
- `345 >> 3` is like `345/2/2/2` and gives 43.

As soon as you need such number magic, you will have to deal with the number representation in the computer again, for example in [Section 4.4.3](#). Until then, remember that C++ has these operations.

So the use of this in the real world is limited to the view of numbers as a series of bits; otherwise `*2` and `/2` make much more sense. So that you can see this use, the following code reverses the bits of an unsigned short.

```
// https://godbolt.org/z/qdY41d5a9
#include <iostream>
#include <bitset>
constexpr unsigned n_bits = sizeof(unsigned short)*8; // 8 bits per char
auto reverse_bits(unsigned val) -> unsigned short {
    unsigned short ret = 0;
    for (unsigned i = 0; i < n_bits; ++i ) {
        ret = (ret << 1) | (val & 1); // one to the side, set lowest bit
        val >>= 1;                    // one in the other direction
    }
    return ret;
}
void tell(unsigned short val) {
    std::bitset<n_bits> bits{val};
    std::cout << val << "=" << bits << "-> ";
    auto lav = reverse_bits(val);
    std::bitset<n_bits> stib{lav};
```

```

    std::cout << lav << "=" << stib << "\n";
}
int main() {
    tell(36u); // Output: 36=0000000000100100 -> 9216=0010010000000000
    tell(199u); // Output: 199=0000000011000111 -> 58112=1110001100000000
    tell(255u); // Output: 255=0000000011111111 -> 65280=1111111100000000
    tell(256u); // Output: 256=0000000100000000 -> 128=0000000010000000
}

```

Operators for Streams

The input and output data streams of the standard library *streams* use the << and >> operators, which are actually intended for bitwise arithmetic, for writing and reading. You have already seen this for `std::cout` and `std::cin` in many listings. In C++ today, << and >> are used far more frequently for streams than for real bitwise arithmetic.

4.3.5 Composite Assignment

If you use an arithmetic operator, a new value is created. If the expression requires intermediate results, these are usually briefly available in the computer memory. Within an expression, new temp-values are then constantly generated and discarded. To avoid this, all arithmetic operators are available in variants that change a variable directly instead.

Instead of `int a = 3; a = (a * 4 + 7 - 3)/4;` you can also write the following:

```

int a = 3;
a *= 4;
a += 7;
a -= 3;
a /= 4;

```

In both cases, `a` then contains the value 4. However, you should avoid such long calculations as the code quickly becomes confusing. Saving the temp-values is only really a big gain in the rarest of cases.

You can also use the operators of bitwise arithmetic in this way:

```

// https://godbolt.org/z/5hqanTeYh
#include <iostream>
#include <bitset> // helps with the output of numbers as a bit sequence
int main() {
    int a = 0;
    for(int idx=0; idx<8; idx++) {
        a <<= 2; // shift two bits to the left: "...100"
        a |= 1;  // set the lowest bit: "...1"
    }
}

```

```
std::cout << std::bitset<16>(a) << "\n"; // 0101010101010101
std::cout << a << "\n";                // 21845
}
```

The *compound assignments* available are therefore:

- The standard arithmetic `+=`, `-=`, `*=`, `/=`, and `%=`
- The binary arithmetic `|=`, `&=`, `^=`, `<<=`, and `>>=`

4.3.6 Post- and Preincrement and Post- and Predecrement

I have already said the most important things about the unary operators `++` and `--`. Let me summarize it once again:

- With `++number`, `number` is first increased by one; with `--number`, it is decreased by one. You can continue to use the result of this calculation in the surrounding expression. Because the operation is performed first here, this is the *pre* variant.
- If you readjust the operator, use the *post* variant. The value of the expression (e.g., `number++`) is then the value of the variable before the change. It is only actually executed at the end of the entire statement. Until then, the computer has to remember the new value somewhere. This may consume memory and time. It is therefore generally better to get into the habit of using the pre variants.
- More importantly, you should never use two of these operators on the same variable within a statement. The compiler allows this, but the result is not defined.

4.3.7 Relational Operators

You can also compare values with each other. Very often you need `==` for equality and `!=` for their opposite. Of course, you can also compare numbers for smaller and larger with `<` and `>` as well as in combinations with equal: `<=` and `>=`.

The result of such a comparison is a true or false outcome—that is, `true` or `false`—and therefore of the type `bool`. These are particularly popular in loops and `if` conditions. However, you can also store the result in a `bool` variable or return it from a function:

```
// as conditions:
if(x < 10) ...
for(int idx=0; idx < 12; ++idx) ...
while(it != end) ...
// store:
bool isLarge = value >= 100;
// return:
bool isPositive(int a) {
    return a > 0;
}
```

With C++20, the *three-way comparison* `<=>` is added. Because of its visual similarity to a spaceship, it is often called the *spaceship operator*. The return of such a comparison is a new (simple) object that can be compared with 0 using `<`, `>`, or `==`, for example:

```
if(a <=> b > 0) ...
```

This corresponds roughly to what `strcmp` does for strings in C or `compareTo` in Java.

This is there to make it easier to provide your own data types with an order. Previously, you had to implement `<`, `>`, `==`, `<=`, `>=`, and `!=` for a new data type if it was to be complete and ordered. As of C++20, it is sufficient to define `operator<=>`, and the compiler derives the normal comparison operators. Implemented correctly, `(a<=>b)>0` is the same as `a>b`, and so on.

4.3.8 Logical Operators

If you want to know whether `x` is between 100 and 200, you must check both relational expressions `x>100` and `x<200`, and both must be true. You could do this with two consecutive ifs. But to combine expressions of type `bool` with each other, there are *logical operators*: they each have two operands and return `bool` again. If the operands `u` and `v` are both `bool` expressions, then

- `u && v` is true if `u` and `v` are both true,
- `u || v` is true if `u` or `v` are true, and
- `!u` is true if `u` is false.

If this is completely new to you, you will find the detailed information in [Table 4.10](#) helpful.

You can then combine the expression:

```
if( x > 100 && x < 200 ) ...
```

Short-Circuit Evaluation

The preceding if statement is in fact equivalent to the following nested if statements:

```
if(x > 100)
    if(x < 200)
        ...
```

Note that the `x < 200` comparison is only evaluated if `x > 100` was actually true. If `x` is 2, for example, then `x < 200` is not reached.

This is important if the second comparison only makes sense (or may only be carried out) if the first was positive. For example:

```
if( y != 0 && x/y > 5 ) ...
```


You certainly know that you can never divide by zero. So if the variable y in x/y has the value 0, then you are doing something forbidden. However, if you check this beforehand, nothing can go wrong. In C++,

- in $u \ \&\& \ v$, the expression v is only evaluated if u is true; and
- in $u \ || \ v$, the expression v is only evaluated if u is false.

This is called *short-circuit evaluation*.

“And” Is Always “And;” “Or” Is Always “Or”

If you define your own types later, you will learn that you can also define the operators on them yourself. This also includes the logical operators.

However, *never* redefine a logical operator in such a way that it does anything other than perform the intuitive calculation. This is because nasty surprises would be inevitable in combination with the short-circuit evaluation: parts of your expression are executed unexpectedly.

If you overload $\&\&$ or $||$ for a custom type, then there is no short-circuit evaluation there. On types other than `bool`, an expression is always evaluated in full.

Alternative Tokens

A small note on the `!` operator: I personally find it a little difficult to see in the source text. Despite its inconspicuousness, it completely reverses the meaning of the whole expression. It's a matter of taste, but I resort to a syntax trick from time to time. You can replace the `!` with the word `not`. For example, `while(!file.eof())...` in Listing 10.1 becomes `while(not file.eof())...`

This is called an *alternative token*, and there are a few of them; the one I find most useful is `not`. I don't want to name them all—not that you'll start using them everywhere. If you are curious, take a look in the language reference under “alternative tokens” or the related area of “di- and trigraphs” (trigraphs are no longer allowed since C++17).

As I said, `not` is not to everyone's taste and you should discuss this with your team. I would advise against the widespread use of these character combinations because they are really very rarely used. Your readers could become confused.

4.3.9 Pointer and Dereference Operators

You have already learned that we call methods with a dot `.`—for example, with `string`:

```
void checkName(std::string& name) {
    if( name.length() ) ...
}
```

You will see later, when we discuss pointers and C-arrays, that you can also use a *pointer* to this variable instead of the reference &:

```
void checkName(std::string* pname) { // Pointer to a string
    if( (*pname).length() ) ...
}
```

If the type of your variable is `string*` and not `string&` or `string`, then it is only “indirectly” linked to the value. To get to the value, use the unary `*` operator—`*pname` in this case. The method call is then `(*pname).length()`. However, as this is somewhat cumbersome, there is the binary `->` operator as a shorter form:

```
void checkName(std::string* pname) {
    if( pname->length() ) ...
}
```

In particular, this is all worth mentioning because both dereference operators (the unary `*` and the binary `->`) can be defined by you on your own types. Knowing them is therefore not only important for pointers and C-arrays.

In fact, pointers are just a very special form of indirection. The standard library is riddled with *iterators*, the generalization of the pointer concept. You will come across iterators in the containers and algorithms and use `*` and `->` there as a matter of course, without pointers being involved. (See Part IV and [Chapter 20](#).)

4.3.10 Special Operators

As we are discussing operators here, two oddballs must also be mentioned.

There is a single ternary (three-character) operator `? :` that allows an if-else query as an expression. It has the following form:

Condition-expression ? Then-expression : Else-expression

Depending on whether the condition is evaluated to true or false, either the then expression or the else expression is the overall result. Note that the two parts must therefore be of the same type so that the compiler can determine the type of the overall expression:

```
// https://godbolt.org/z/xrMYz93WP
int main() {
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 0..9
            int max = w1 > w2 ? w1 : w2; // ternary operator
        }
    }
}
```

Here `max` is assigned the larger of the two values `w1` and `w2`.

The comma can be used as a *sequence operator* in expressions. If you write several expressions in parentheses (...) and separate them with commas, then the compiler evaluates the expressions from left to right, but only keeps the result of the last expression as the overall result. The construct therefore only makes sense if the leading expressions have side effects.

It looks like this for two expressions:

(Expression-1, Expression-2)

The compiler first calculates *Expression-1* and then *Expression-2*. The result of the first expression is dissipated so that the total expression receives the value and type of the second expression.

```
// https://godbolt.org/z/rq66cWr3d
int main() {
    int a = 0;
    int b = 0;
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 0..9
            int max = w1 > w2 ? (a+=b, w1) : (b+=1, w2); // sequence operator
        }
    }
}
```

Listing 4.26 You can concatenate several expressions with commas in brackets.

If $w1 > w2$, then the expression $(a+=b, w1)$ is evaluated. Although $w1$ is returned as the result for `max`, the expression $a+=b$ is executed first. In the case of $w \leq w2$, the term $w2$ from $(b+=1, w2)$ is assigned to the variable `max` after $b+=1$ has been evaluated.

I have deliberately chosen a particularly bizarre example here for the sequence operator because, first, it is difficult to find a meaningful example; and, second, you should avoid this special construct like Darth Vader avoids magma. The sequence operator thrives on side effects, but within an expression, each variable must not be affected by more than one side effect, so its use is dangerous.

There are exceptions where it makes sense to use it—namely, where only a single expression is permitted and it would otherwise be more complicated.

This can be the case, for example, in the incrementation part of `for` loops.

```
// https://godbolt.org/z/vd85f7rMj
#include <iostream>
int main() {
    int arr[] = { 8,3,7,3,11,999,5,6,7 };
    int len = 9;
```

```

for(int i=0, *p=arr; i<len && *p!=999; ++i, ++p) { // first ++i, then ++p
    std::cout << i << ":" << *p << " ";
}
std::cout << "\n";
// output: 0:8 1:3 2:7 3:3 4:11
}

```

Listing 4.27 The sequence comma can be useful in for loops.

Note that in a declaration `int a, b, c;`, the comma is not the sequence operator. This also applies to the declaration part of the `for` loop, `int i=0, *p=arr`, where the comma only separates the declarations.

The comma that separates function arguments in `func(x,y,z)` or list elements in `{1,2,3}` is not the sequence operator either.

4.3.11 Function-Like Operators

Even if they do not look like it, the following cases also belong to the operators:

- ***kind*...*cast*<target_type>(value)**

All of these are *type conversions*. `value` is converted to `target_type` in the C++ notation. There are multiple different kinds: `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. They differ slightly in what they do and what conversions they can perform, but the type of the expression is always `target_type`. If a type conversion is not possible, the compiler reports an error. Otherwise, the `value` is made to fit, which harbors dangers. For example, when converting from `long` to `short`, information can be lost without you realizing it. Forcing types is not desirable, but it does happen sometimes. In an optimal world, you can manage without explicit type conversions.

- **(type)value**

This is the type conversion notation adopted from C. In `(int)value`, the compiler attempts to convert `value` into an `int`, regardless of the type of `value`. Depending on the context, this notation corresponds to one of the C++ notations, most commonly `static_cast<>`. Simply because you are emphasizing the type of conversion, you should always choose the C++ notation.

- **sizeof**

You can use `sizeof(type)` and `sizeof(value)` to find out the size of a type or a variable in `char` units. A `sizeof(char)` always returns 1.

- **new and delete**

`new class{}` and `delete var` are used to manage dynamic memory, as you will see in [Chapter 20](#).

- **throw**

With `throw ExceptionClass{};` you trigger an exception, which is explained in [Chapter 10](#).

4.3.12 Operator Sequence

If you use several operators in an expression—possibly different ones—then these are evaluated in a specific order.

Just as you can demand that a proper pocket calculator observes the rule of *point before dash calculation* for $3+4*5+6$ and produces the correct result of 29, C++ also masters this. Furthermore, all other operators also have a *precedence*: the higher an operator is in this ranking, the earlier it is evaluated compared to other operators.

Remember this simple sequence, which starts with the strongest bond:

- **Multiplicative**

`*`, `/` and `%`

- **Additive**

`+` and `-`

- **Stream operators**

`<<` and `>>`

- **Compare**

`<`, `<=`, `>` and `>=`

- **Equality**

`==` and `!=`

- **Logical**

In this order: `&&` then `||`

- **Assignments with =**

But also all compound assignments such as `+=`

This allows you to write many a complex expression without having to correct the meaning with parentheses:

```
bool yesno = 3*4 > 2*6 && 10/2 < 13%8;
```

This saves you the parentheses shown in the following:

```
bool yesno = (((3*4) > (2*6)) && ((10/2) < (13%8)));
```

Note that the stream output operator `<<` binds more loosely than normal arithmetic with `+` and `*`. However, if you have comparisons in the output, you need brackets around the comparison:

```
std::cout << 2*7 << x+1 << n/3-m << "\n"; // no () required between <<
std::cout << (x0 >= x1) << (a<b || b<c); // (): << would bind more tightly
```

But what happens if several operators of the same precedence are next to each other? In expressions such as $10-5-2$, the left $-$ is evaluated first because $-$ is *left-associative*—as if the expression $((10-5)-3)$ were in parentheses. The result is therefore 2. All binary arithmetic, Boolean, and comparative operators are left-associative, so you can intuitively calculate with them. Sometimes this is also called *left-to-right associative* or *LR-associative*.

The group of assignment operators, in turn, is consistently *right-associative*, which is why the compiler for $x += y += z += 1$ executes $(x += (y += (z += 1)))$ as expected and first increments z by 1 in order to then deal with the other variables one after the other. If the expression were evaluated as $((x += y) += z) += 1$, then x would receive a new value several times because $x += y$ also returns x , and then $+= z$ would be executed; y would not be changed. And x would be returned once again, and $+= 1$ would increment it by 1 instead of increasing z .

Most of the time, precedence and associativity work intuitively and as expected. If in doubt, it is better to bracket as later readers will probably ask themselves the same questions as you.

4.4 Built-In Data Types

So far you have only learned about types in a flyover—and around them, just as much fundamental C++ as was necessary to understand them.

This section builds on these basics. You now know enough about the language to understand the comprehensive explanations of all the built-in types.

In this section, you will learn which data types are available to you in C++ if you do not use `#include` in your program or do not use the *standard library*. What are the *built-in data types*, and what can you do with them?

Pay attention to the types of variables and function parameters in the following example.

```
// https://godbolt.org/z/MPPsbdq9c
#include <iostream> // cin, cout for input and output

void input(unsigned &birthDay_,
           unsigned &birthMonth_,
           unsigned &birthYear_,
           unsigned long long &taxnumber_,
           double &bodyheight_)
{
    /* Inputs still without good error handling... */
    std::cout << "Date of birth: "; std::cin >> birthDay_;
    std::cout << "Month of birth: "; std::cin >> birthMonth_;
```

```
std::cout << "Year of birth: "; std::cin >> birthYear_;
std::cout << "Tax number: "; std::cin >> taxnumber_;
std::cout << "Height: "; std::cin >> bodyheight_;
}

int main() {
    /* data */
    unsigned birthDay_ = 0;
    unsigned birthMonth_ = 0;
    unsigned birthYear_ = 0;
    unsigned long long taxnumber_ = 0;
    double bodyheight_ = 0.0;
    /* Input */
    input(birthDay_, birthMonth_, birthYear_, taxnumber_, bodyheight_);
    /* Calculations */
    // ...
}
```

Listing 4.28 Some new data types are used here.

I have used the following data types:

- `unsigned` and `unsigned long long` as representatives of the *integer data types*
- `double` for saving a (floating-) point number
- The variables `std::cin` and `std::cout`, whose types are not explicitly mentioned

The first important distinction between these data types is whether they are built-in or not.

4.4.1 Overview

If you do not use `#include`, you have a very limited number of data types available. For example, you can define additional types with `class` or create aliases with `typedef`, but your selection is then very limited:

- **Integers—`int`, `short`, `long`, and `long long`, each signed or unsigned**
Integers store numbers without a decimal point—for example, 3, -12, and 987654321. The different-sized variants store either signed or unsigned numbers. Consider which variant you need for each application. When in doubt, `int` is a good choice. Once you have chosen the type, you must ensure above all that you adhere to the number range of the type in order to avoid a dangerous *overflow*.
- **Floating-point numbers—`double`, `float`, and `long double`**
Integers are unsuitable for some things. Use `double` to store decimal numbers such as 3.14, -0.00001, or 6.281e+26. Their space is still limited, even if it is large, but the

biggest problem is the *accuracy*; you cannot store even a value of 1/3 accurately. Always use floating-point numbers with caution.

- **Truth values—`bool`**

`bool` is a data type for storing truth values. A `bool` is either `true` or `false`. You have often used this type in `if` statements and the like. Although a single bit would actually suffice to store `bool`, this data type is by no means more space-saving than `char`.

- **Character types—`char`, `char16_t`, `char32_t`, and `char8_t`**

In C++, there are only a few places where there is a real distinction between numbers and characters, so you can also use the `char` character type as a number. This is the smallest unit, and if you ask for the actual size of a type or a variable with `sizeof(x)`, then it is the number of `char` units that is returned to you. `char` becomes a character type because you store character strings in `char` sequences—either as an array in `char[]` or in a string from the standard library. As `char` is not designed for international characters (Unicode), there are `char16_t`, `char32_t`, and the somewhat outdated `wchar_t` with more space. Examples of characters are `'a'`, `'Z'`, and international (Unicode) characters. In C++20, `char8_t` is added.

- **References—`&`**

Indirect references with `&` refer to another variable. This allows you to address the same variable under a different name—for example, `int x = 7; int& ref = x;`. Changes to the reference, such as `ref=12`, actually change the original—in this case, `x`. As long as it exists, a reference always refers to the same variable. A reference cannot refer to “nothing.”

- **Pointers and C-arrays—`*` and `[]`**

Indirections with `*` point to a variable as a memory address, in the case of a C-array as `[]` with length information. A character string such as “Your name” is an example of `char[9]`⁶ and is passed to functions as `char*`. We will discuss this in [Chapter 20](#) on pointers. In contrast to references with `&`, pointers can be changed so that they refer to another variable. If you refer to `nullptr`, this stands for the reference to “nothing.”

In the source code, data is represented by variables (or similar) or by literals. When the compiled program is running, however, things look quite different. Then the data needs a physical representation: in the processor, where it is calculated, and in the memory, where it is loaded, saved, and changed. Memory can mean many things here: from the fast register to the cache, the main memory and graphics card memory to the hard disk, and, in a broader sense, the network or cloud.

Each data type has a physical representation that the processor understands. In this section, we will briefly discuss this representation, because C++ is also a language that is close to this representation and can manipulate it directly.

⁶ Such character strings are internally terminated with an end character, which is why the array length is 9 and not 8, like the visible characters.

4.4.2 Initialize Built-In Data Types

One of the most important differences between these data types and the majority of other—non-built-in—data types is that you *have to initialize* the variables of these types! For example, write `int x = 7;` or `int x{7};`. If you omit the initialization with 7, then `x` is not initialized at all. Its initial value is then random, which can have consequences.

If you want to leave it not to chance but to the compiler to choose a “good” value for the initialization, then write a pair of empty curly braces `{}` after the variable. The rule of thumb is that it is then initialized with zero or something equivalent:

```
int x{};           // initializes x with 0 - built-in type
double y{};        // y becomes 0.0 - built-in type
std::string s{};    // empty string - class
struct tm mytm{};   // all fields 0 - tm is aggregate from <time.h>
```

Value Initialization Leaves Nothing to Chance

You can get into the habit of always initializing at least with `{}` if you have no other more sensible initialization, because even the noninstalled types benefit from this. So you are not doing anything wrong if you include this in your repertoire. It is called *value initialization*, which means initialization with a *meaningful* type-specific value. For types that don't provide anything else with `{}`, you at least get something other than cosmic noise in the computer memory.

Among the noninstalled types, there are a few exceptions that do not tolerate initialization with `{}`. The compiler will point these out to you. These are usually classes where you have to specify arguments between the braces.

4.4.3 Integers

We have used the `unsigned` type in several places in the examples. This is just an abbreviation of `unsigned int`. In contrast to the normal `signed int`, abbreviated `int`, this can store slightly larger positive numbers, but not negative ones.

Normally, you should use the `int` type for integers. However, the number range is limited. If you need more length, you can use `long` and `long long`. If you need to save space, there is `short`. All these variants are also available in the `unsigned` variant, such as `unsigned long`. It is not necessary to write `signed` in front of the number type instead as this is the default. An `int` (whether `signed` or `unsigned`) is what your system handles most naturally and can usually calculate the fastest.

You can perform arithmetic calculations with the integer types—that is, multiplication and the like. However, you are already reaching the limits of the data type with basic

arithmetic operations, because logically an unsigned int cannot take on a negative value. And both the signed and unsigned variants have a limit for large numbers.

To understand what actually happens when calculating with int types and relatives, you only need to know that the computer works in the *binary system*. This is nothing other than the *decimal system* that you are used to, except that each digit cannot be 0 to 9, but only 0 or 1—one of two possible values (hence the *bi* in *binary*). In [Section 4.3](#), I discuss this in the “Binary System” box.

The individual positions that can be 0 or 1 are called *bits*. How many bits fit into a variable of an integer type depends on your system. The standard says that char is the smallest and long long is the largest type, with short, int, and long in between, in that order. On today's machines, char is eight bits and long long is 64 bits. [Table 4.3](#) gives examples of two common systems today. The Windows column applies to both 32-bit and 64-bit Windows, as well as 32-bit Linux.

Bits	Linux, 64-Bit	Windows	Minimum For...
8	char	char	char
16	short	short	short, int
32	int	int, long	long
64	long, long long	long long	long long

Table 4.3 Bit widths of two architectures and the minimums as examples.

If an unsigned int has 32 bits, then the largest number that this data type can represent is $1 \cdot 2^{31} + 1 \cdot 2^{30} + \dots + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^{32} - 1$, which corresponds to just over 4 billion. This is shown in [Table 4.4](#). You must use your program to ensure that an expression of type int never assumes a larger value.

Minimum Widths for Integer Types

Since C++20, the numbers listed under “Bits” apply as the minimum widths of the implementation for the data types specified in the “Minimum For...” column. However, you don't have to get used to this, as all implementations actually adhered to these widths even before standardization. However, you will probably still find people for a long time who believe that C++ does not specify minimum widths except for char. Now you can shine with new knowledge.

A note on the names of the integer types: officially, the types are called int, unsigned int, long int, and long long int. I myself usually abbreviate unsigned int to unsigned, but you will often see it written out in full. With long and long long, the written-out version with int behind it is rarely used.

Bits	Unsigned	Signed
8	0 ... 255	−128 ... 127
16	0 ... 65 535	−32 768 ... 32 767
32	0 ... 4 294 967 295	−2 147 483 648 ... 2 147 483 647
64	0 ... 18 446 744 073 709 551 615	−9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807

Table 4.4 Numerical ranges.

Physical Representation of Integers

Have you noticed that the number of bits that can be integers wide is divisible by eight? I'm probably not telling you anything new by saying that in most computer systems, eight bits make up one *byte* (there are exceptions). In C++, a byte is the smallest *addressable* data unit. A byte in memory has an address x ; the byte next to it has address $x+1$.

On a typical system, an `unsigned int` has 32 bits—that is, four bytes, as you can see in [Table 4.3](#). The unit that is particularly advantageous on a given architecture is called a *data word* (or simply *word*). For the purposes of this book, let's assume that four bytes make up a data word. However, this need not be the case in all architectures.

From this we derive the terms *half word*, *double word*, and *quadruple word*, which comprise 16 bits, 64 bits (eight bytes), and 128 bits (16 bytes). If you assume a smaller word size, there is also the term *eightfold* or *double quad word*. Instead of *half word* or *double word*, *short word* or *long word* is often used, and here you can see the proximity of C and C++ to machine representation. However, particular caution is required, as Linux and Windows jargon diverge here; *long word* in particular can sometimes mean the same thing as *word*.

Last but not least, there is the term *nibble*, which refers to half a byte: the upper nibble denotes the four high-order bits, the lower nibble the four low-order bits. You can represent a nibble with a hexadecimal character (see [Table 4.7](#)).

As already mentioned, the terminology may be shifted on different architectures. However, it is important to know the terms because they are often used in documentation. Interface descriptions for specialized hardware such as graphics cards refer to them (and often have their own interpretation of the widths). Hardware-related tools also refer to these terms. For example, in assembly code (AT&T or GAS syntax) on an x86-64 Linux, `imull` means *multiply long* and is used when a C++-`int` or `-short` is multiplied. An `imulq` for *multiply quad* multiplies `long` or `long long`. The same code on a 64-bit MIPS uses `mul` for `short` and `int` and `dmult` for `long` and `long long`. What I mean by this is that you always have to be careful about what the tool you are using in your architecture means. In [Table 4.5](#), I have compiled the same code in assembler on two different 64-bit architectures. Note the suffixes of the commands that refer to the operand width.

C++	x86-64-Linux	MIPS64
<pre>long long sq(long long n) { return n * n; }</pre>	<pre>sq(long long): imulq %rdi, %di movl %rdi, %rax ret</pre>	<pre>sq(long long): dmult \$4,\$4 j \$31 mflo \$2</pre>
<pre>long sq(long n) { return n * n; }</pre>	<pre>sq(long): imulq %rdi, %di movq %rdi, %rax ret</pre>	<pre>sq(long): dmult \$4,\$4 j \$31 mflo \$2</pre>
<pre>int sq(int n) { return n * n; }</pre>	<pre>sq(int): imull %edi, %di movl %edi, %eax ret</pre>	<pre>sq(int): j \$31 mul \$2,\$4,\$4</pre>
<pre>short sq(short n) { return n * n; }</pre>	<pre>sq(short): imull %edi, %di movl %edi, %eax ret</pre>	<pre>sq(short): andi \$4,\$4,0xffff mul \$2,\$4,\$4 j \$31 seh \$2,\$2</pre>
<pre>char sq(char n) { return n * n; }</pre>	<pre>sq(char): movl %edi, %eax imull %edi, %eax ret</pre>	<pre>sq(char): andi \$4,\$4,0xff mul \$2,\$4,\$4 j \$31 seb \$2,\$2</pre>

Table 4.5 Even on two 64-bit architectures, different tools can select different terms for word sizes.

Whether words or bytes, they are stored next to each other in the memory. Assuming an unsigned int has 32 bits, the abstract integer 123456789 is stored in four bytes with the values 7, 91, 205, and 21. You can recalculate the original value from this by interpreting the individual values as positions in a 256 number system—that is, $7 \cdot 256^3 + 91 \cdot 256^2 + 205 \cdot 256^1 + 21 \cdot 256^0$.

The values of bytes are rarely written as decimal numbers. It is very common to write them as hexadecimal numbers—that is, 0x07, 0x5b, 0xcd, and 0x15. This means that each byte has exactly two letters (plus the 0x prefix). If you add them together, you get an eight-character hexadecimal number: 0x075bcd15 or, without a leading zero, 0x75bcd15.

With this notation, you can now beautifully represent a densely packed sequence of int values in memory (see [Figure 4.2](#)).

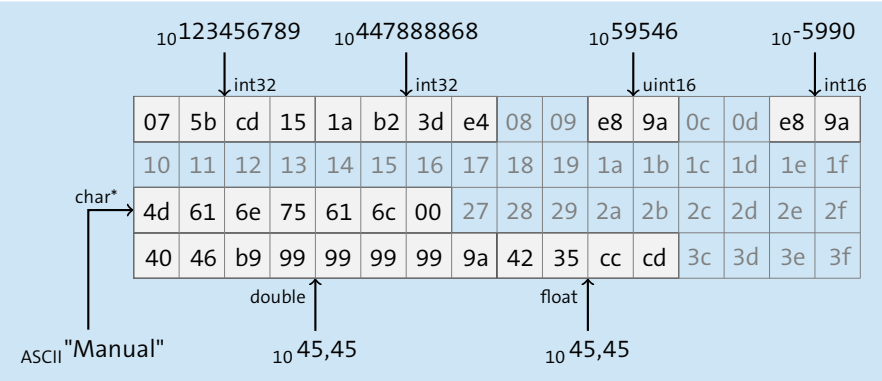


Figure 4.2 Data in memory (big-endian).

The two decimal numbers 123456789 and 447888868 at the beginning are stored directly next to each other. The content of the memory cells is shown in hexadecimal numbers.

The figure contains further data for illustration purposes. The hexadecimal byte sequence 0xe89a is interpreted as 59546 for a `uint16_t` (e.g., unsigned short). If the byte sequence is read with an `int16_t` (e.g., short), this corresponds to -5990. The C string "Manual" has a terminating `\0` byte. Which letters mean which hexadecimal byte is defined in the *ASCII standard*. Finally, I have stored the decimal floating-point number 45.45 as `double` and `float`, for which you see the hexadecimal representations.

Memory Alignment

It is not entirely coincidental that the data elements in the figure are always stored such that their first byte is at an address divisible by four. This is called *memory alignment*. Four bytes correspond to a 32-bit alignment. Most modern machines work faster when this is the case. To achieve optimum speed, data is not packed close together if necessary, but shifted with “empty” bytes in between. In the example, this is the case at 59546 and -5990: the addresses 0c and 0d in between do not contain any meaningful data.

Some machines cannot even cope if the data of certain data types is not stored with the correct alignment. In this case, the data must first be copied in a time-consuming process when reading. In addition to 32-bit alignment, there is also 16-bit, 64-bit, and even 128-bit alignment. No alignment means that the start position does not matter; this corresponds to 8-bit alignment and occurs in practice as well.

You do not normally have to worry about memory alignment; the compiler does this for you.

Byte Sequence

For the rest of this section, I would like to omit the `0x` prefix. If not mentioned otherwise, decimal numbers like 123 are set in normal font, hexadecimal numbers like `a87f` in listing font.

The number 447888596 is divided into bytes—that is, `1a`, `b2`, `3c`, and `d4`. At address `x` you store `1a`, at address `x+1` `b2`, at `x+2` `3c`, and finally at `x+3` `d4`—or written together, at addresses `x..x+3` `1a b2 3c d4`. In [Figure 4.3](#), I have illustrated this. If you number the eight hex digits from their lowest significance so that the digit with the highest significance is given the number 8 and the one with the lowest the number 1, the numbering is `87 65 43 21`.

This format seems natural to us because, like our number system, it names the most significant digit of the whole number first and the least significant last—just as we assign the highest significance to the digit 1 and the lowest to the digit 3 in 123. This memory format is called *big-endian*. It stores the *most significant byte* (MSB) first.

The other way to store the number 447888596 is the *little-endian* memory format—with the least significant element first. This then results in the values `d4 3c b2 1a` at the addresses `x..x+3` in the memory—so, with the *least significant byte* (LSB) first. In my opinion, this looks confusing, especially because each individual two-digit hex number nevertheless names its larger component first and the lower component afterward. If you number the eight hex digits again as before, the result is `21 43 65 87`. It helps me to write down such a sequence the other way round.

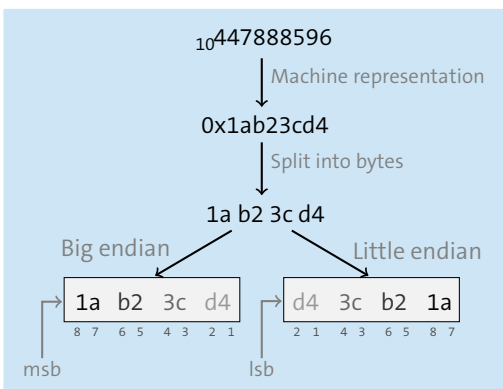


Figure 4.3 Numbers can be stored with the highest or lowest byte first.

However, LSB also has advantages. If the computer starts reading at address `x` and interprets the byte there, the significance of `d4`, for example, which it reads, always remains the same, regardless of whether `d4` is part of an `int`, a `short`, or a `long`. It is a small advantage that the early CPUs probably made use of.

Why am I asking these questions and not simply dismissing little-endian as an esoteric eccentric? Because it isn't one: Intel processors, including the Pentium inside your

computer—I claim, without taking any great risk of being wrong—is one of these representatives. Of course, this also applies to the compatible AMD Opterons and accelerated processing units (APUs). By the way, another name for this memory format is *Intel byte order convention*. But other early acquaintances were also born as little-endian: the famous Zilog Z80 processor, as well as the honorable MOS Technology 6502 of the Commodore 64, Apple II, Atari, or NES.

What has historically competed against it? Big-endian is spoken of in the Motorola 68k family, PowerPCs, MIPS, and SPARC. Yes, a few of them are still around.

Cross-Platform Presentation

To prevent the data from being mixed up due to different interpretations of the byte order by the sender and receiver when exchanging data between machines, there is an agreement, particularly in the network area, on the byte order in which the data is to be transmitted. This is therefore called the *network byte order*. When you write a program, your program is the “host,” so you are always responsible for translating the data from the host order to the network order when saving or sending and vice versa when reading or receiving.

Most platforms have functions such as `htons` (host-to-network short) and `ntohl` (network-to-host long). You should always use these functions if you want to write a portable program. However, read the specification of your protocol carefully as there are exceptions.

Are you now asking whether the network sequence is little- or big-endian? Do you really want me to tell you? If you want to write portable programs, you shouldn't need to know; you should use the conversion functions even if the host and network order are identical. But if you really want to know: the network order corresponds to big-endian. So if you are on such a system, the functions will do nothing. Better call them anyway.

With the internet, the big-endian camp has gained a strong representative. And this is perhaps also the reason that the current conquerors—the mobile CPUs of cell phones, tablets, and other connected devices, in which an ARM computing unit usually performs its services—are *bi-endian*; that is, they can do both.

Integer Literals

In the source code, you can simply write down an integer in the decimal system. You can use a suffix to specify whether it is `signed` or `unsigned` and what width the literal represents. Without a suffix, it is an `int`. In practice, however, this specification is only necessary in special cases as the compiler itself recognizes what type of integer the literal is during initialization and the like. In [Table 4.6](#), you can see the possible suffixes with examples.

Type	Suffixes	Examples
int		42, -12
unsigned	u, U	3u, 123U
long	l, L	-999l, 88'888L
unsigned long	ul, UL,	77'777ul, 6'666UL
long long	ll, LL	43'214'321ll, -12'341'234LL
unsigned long long	ull, ULL	23'443'212ull, 1ULL

Table 4.6 Integer suffixes and the corresponding types.

As of C++23, you can also use the suffix `z` or `Z` for a `size_t` literal, optionally combined with `u` or `U`—for example, `42z` or `42zu`. This is useful, for example, if you compare a literal with the return value of the `size()` function, such as `(10z > size(v))`.

Since C++14, you can place single quotation marks `'` between digits in any number literal. They do not affect the number. For decimal numbers, groups of three are useful to achieve the usual reading grouping.

C++ also allows you to write an integer in the base 8 system (*octal*) or 16-digit system (*hexadecimal*). These have the advantage that three or two digits always result in a byte (eight bits) and are sometimes easier to read than in the decimal system. Since C++14, you can even write a number in the base 2 system (*binary*) as a literal. To specify the number system (the base) of the literal, use a prefix. In [Table 4.7](#), you can see the possible prefixes. Both the prefixes and the letters of the 16-letter system are case-insensitive; for example, no distinction is made between `0xFF` and `0xff`.

Here too, you can insert single quotation marks `'` between the digits to group them. Groups of three are common for decimal and octal literals. Groups of two, four, or eight are common for binary and hexadecimal literals.

It becomes more advanced if you combine the prefixes for the number system with the suffixes of the data type—but as I said, the suffixes are rarely necessary. With `0xffULL`, you define 255 as an unsigned long long.

Base	Prefix	Examples	Decimal as int
10		255, -4	255, -4
16	0x	0xff, 0xCAFE	255, 51966
8	0	0377, 0777666	255, 262070
2	0b	0b1111'1111, 0b0101	255, 5

Table 4.7 Integer prefixes for the different number systems.

Operations on Integers

You can calculate arithmetically with all integer types, as you would expect: multiplication with `*`, addition and subtraction with `+` and `-`, and division without remainder with `/`. The latter means that if you calculate `20 / 7`, the result is 2; the decimal places are simply cut off. With `%`, you get the remainder (*modulo operation*).

```
// https://godbolt.org/z/ebPcdxvj
#include <iostream>
int main() {
    std::cout << 3 + 4 * 5 + 6 << "\n";           // 29
    std::cout << 20/7 << " Rest " << 20%7 << "\n"; // 2 remainder 6
}
```

Listing 4.29 Arithmetic with integers.

For more complex calculations, you must write your own functions or use floating-point numbers such as `double`, for which there are a large number of calculating functions in the standard library in the `<cmath>` header.

The bit operations have already been described in [Section 4.3](#). The following listing is another short example.

```
// https://godbolt.org/z/jeTs5Exqx
#include <iostream>
int main() {
    unsigned a = 0b1111'0000; // 240
    unsigned b = 0b0011'1100; // 60
    std::cout << ( a | b ) << "\n"; // Bit-Or: 252, in bits 1111'1100
    std::cout << ( a & b ) << "\n"; // Bit-And: 48, in bits 0011'0000
    std::cout << ( a ^ b ) << "\n"; // Exclusive-Or: 204, in bits 1100'1100
    unsigned int c = 170; // in bits 0...(24x0)..0'1010'1010
    std::cout << ( ~c ) << "\n"; // Inv.: 4294967125, Bits: 1...(24x1)..1'0101'0101
}
```

Listing 4.30 Bit operations.

At this point, it is important to me that you understand the concepts of integers in C++ and learn how to use them. I will point out the typical pitfalls and give a few case studies:

- There are the integer types `char`, `short`, `int`, `long`, and `long long`, each in signed and unsigned form.
- These have different widths on the different systems; only the order of their sizes is fixed.
- `char` is special because it is used not only for arithmetic but also for storing characters (hence the name).

- Your general-purpose integer types should be `int` and `unsigned`.
- You can calculate “normally” with all numbers in C++; simple arithmetic and bit operations are supported directly.
- Within an expression with two operands of different types, one may be converted before the calculation is executed.

Integer Overflow

If you calculate `unsigned int x = 5 - 10;`, then strange things happen: for the computer, this becomes a huge number. The same thing happens at the other end of the spectrum: on many current computers, an `unsigned int` has 32 bits, and if these are not enough to store a number, the computer starts again from zero. Both are called an *overflow*. Be careful here if you are working with the `signed` variants, as these do not specify what happens in the event of an overflow. In general, an overflow should be avoided unless you know exactly what you are doing with a `signed` type.⁷

Arithmetic Type Conversion

So far, we have made it easy for ourselves: we have always given an operator two operands of the same type. If you use two different types, the compiler performs an *implicit type conversion* for the smaller integer type: it is first copied into a temporary variable of the larger type. The result type of the expression is then the larger type.

For example, with `unsigned char c = 50;` `int n = 500;` in the expression `c+n`, the `c` is converted into an `int` with the value 50 before the calculation. This makes it possible for the result of 550 to be calculated at all (a `char` would be too small for this). The result type of the expression is also `int`, as this is the larger of the two types involved.

It gets tricky when you mix `signed` and `unsigned` types. Here are a couple of rules of thumb:

- If the `signed` operand involved is larger than the `unsigned` operand, it will also be used for the result, and you can be reasonably sure that the conversion will not destroy anything.
- If both are the same size, the result is given the `unsigned` type and any negative numbers are lost.

In practice, this has the following effect: `signed` and `unsigned` are both the same size—both `int`. If you add a `signed` `ss = -4;` and an `unsigned` `uu = 2`, the compiler specifies `unsigned` as the type for the result. This has the effect that `-4` is converted to `unsigned` before the addition—and thus becomes 4294967292 (for `int` with 32 bits). The result is then somewhat surprisingly 4294967294. And as these rules apply to comparisons as well as arithmetic, `if(ss < uu)` is also surprisingly evaluated to `false`—or even more

⁷ *How Disastrous Is Integer Overflow in C++?*, <http://stackoverflow.com/questions/9024826>, [2014-02-02]

explicitly, the value of the expression `if(-4 < 2u)` is determined as `false` by the compiler in C++ because the signed literal `-4` is compared with the unsigned literal `2u`.

Avoid Mixing

If you are dealing with number ranges that are at risk during such a conversion—negative or large numbers—then you should not rely on the implicit conversion. In this case, convert the operands yourself beforehand. Do not write `if(mySize > theLimit)...` if the two types have different signs, but convert them yourself. This will make you aware that something is happening here that you need to keep an eye on:

```
if(mySize > (long)theLimit) ...
```

Better still, both figures would have been the same type right from the start.

As of C++20, safer auxiliary functions such as `std::cmp_less()`, `std::cmp_equal()`, and the like are available, but these are more complicated to use than simply writing `<` or `==`. To check whether a value can be safely stored in a specific type, use `in_range<T>()`. In generic template code, this is often the only safe way.

Alias Number Types

There are number types that are defined differently on systems. A prominent example is `size_t`: sometimes this type is identical to unsigned `int`, sometimes to unsigned `long` or even to unsigned `long long`. But if you only use variables of these types for the purpose for which they are intended, you are not interested in the exact underlying type.

For example, if you want to know the size of a data structure, use `sizeof(data)`. The return is a `size_t`—and you should store the result in this. Of course, this is actually one of the other `int` types, but you don't need to know exactly which one. It is an *alias* for the (unknown) real type.

There are several reasons to use an alias. The most important are these:

- You save yourself typing work. An alias is shorter, especially with templates.
- They say something about the usage; that is, they use the alias as a *semantic label*. `ptr_diff_t` and `size_t` may be identical, but their names already tell you something about their usage.
- They are more flexible for later changes, as different types may be useful for specific purposes in different systems. In this case, you only need to change the alias in one place.

Aliases such as `size_t` have been created with `using` and are usually available if you are already using the corresponding `#include` anyway. For example, `size_t` could have been defined like this:

```
using size_t = unsigned long long;
```

“typedef” versus “using”

If you look now, you might not find any `using` in your headers at all, because the way to do this before C++11 was as follows:

```
typedef unsigned long long size_t;
```

I recommend that you use the new style. It may be a matter of taste (or habit), but I personally like the style where the newly introduced name is to the left of an equals sign `=`. But I'm showing you the `typedef` style because we'll probably be seeing it for a very long time.

As you will see in Part IV, a `vector` can hold a large number of elements. You can access the individual elements by index using square brackets `[]`. This index must be a number—more precisely, a `size_t`. So if you define a count variable for a vector index, you can very well let it be a `size_t`.

```
// https://godbolt.org/z/Gso3xGcj
#include <vector>
#include <cstdint> // size_t
int main()
{
    std::vector<int> data = { 100, -4, 6'699, 88, 0, } ;

    int sum = 0;

    for(size_t idx = 0; idx < data.size(); ++idx)
    { // a specific int type
        sum += data[idx];
    }
}
```

Listing 4.31 Index variables can be of type “`size_t`”.

However, it should not be concealed from you that the sum in the preceding example could of course have been implemented much better using `for(auto d: data) sum += d;`. Or you could immediately use an algorithm from `<algorithm>` and `auto sum = std::accumulate(data.begin(), data.end(), 0);`. In both cases, you can see that with C++11 you can now often save yourself the trouble of knowing the exact type.

Other frequently used aliases with typical definitions for certain integer types are shown in [Table 4.8](#).

Alias	On 64-Bit Linux	Use
size_t	long unsigned int	Sizes that cannot be negative, such as container sizes
ptrdiff_t	long int	Distance between two addresses as <code>ptrdiff_t d = &x - &y;</code>
time_t	long int	From <code><ctime></code> to store seconds since 1.1.1970
int8_t	signed char	Integer with exactly eight bits from <code><stdint></code>
int64_t	long int	Integer with exactly 64 bits from <code><stdint></code>
uint8_t	unsigned char	Positive integer with exactly eight bits
int_least64_t	long int	Integer with at least 64 bits
int_fast16_t	long int	Fast integer with at least 16 bits
uint8_t	unsigned char	Positive integer with exactly eight bits
byte	<i>special</i>	Access and manipulate raw memory with <code><cstdint></code>

Table 4.8 Some frequently used aliases for certain integer types.

All types from `<stdint>` are available for 8, 16, 32, and 64 bits. The least and fast variants can be larger. However, the `int8_t` to `int64_t` variants are only defined if they exist on the system in exactly this size—which should usually be the case.

std::byte as a Special Case

Traditionally, C and C++ programmers often prefer to use one of the integer types to manipulate raw memory. This is actually not ideal. Unless you want to do arithmetic like `+` or `*` with the values, `int` or `unsigned char` can actually do too much. In most cases, you only need to perform bit manipulation (`|`, `&`, `<<`, etc.). And this is exactly what `std::byte` from the `<cstdint>` header was added for in C++17.

This data type is intended for manipulating raw memory and has the width of an `unsigned char`, which means eight bits on the most common platforms. If you have two variables, `byte a` and `byte b`, then the following operations are available to you, where `n` stands for any integer value:

- `std::byte b{n}` creates a new byte.
- `to_integer(b)` returns the numerical value.
- `b << n, b >> n` returns a new byte with bits shifted to the left or right.
- `b <<= n, b >>= n` shifts bits in `b` to the left or right.
- `a | b, a & b, a ^ b` links two bytes bit by bit using OR, AND, or exclusive-or (XOR).

- `b |= a`, `b &= a`, `b ^= a` saves the result of the link back to `b`.
- `~b` returns a new byte with all bits negated.

A typical use is `vector<byte>`, in which you then read the content of a file, for example.

4.4.4 Floating-Point Numbers

In addition to the integer types, there are also `float`, `double`, and `long double` types for floating-point numbers. Instead of throwing away the remainder of $5/2$ when dividing, you can store it in a `double`; for example, `double x = 5.0 / 2.0;` stores 2.5 in `x`. But beware, floating-point numbers also have their limits: they also only have a fixed number of bits available, but they use them differently than the integer types do. They use some of the bits for “scaling” the value. This means that you can use them to store extremely large and tiny numbers, but only up to a certain precision. You need to keep the following things in mind:

- **Overflow**

Although the number range is much larger than for `int`, a `double` can also overflow. It cannot store numbers larger than 10^{300} and smaller than -10^{300} .

- **Precision**

Precision is much more critical. The further you move away from zero as the center, the coarser the values that `double` can store become. The distance between two “neighboring” `double` numbers increases. From around 10^{15} , this is more than 1. If you store them in `double`, the numbers $10^{16}+1$ and $10^{16}+2$ are identical for the computer; see [Listing 4.33](#).

- **Rounding**

But `double` also has problems with numbers close to zero. Because where an `int`, for example, can store 7 exactly, a `double` has problems with 0.1—and even more so with $1/3$, so 0.3, as you can see in [Figure 4.4](#).

The standard data type is `double`. The bit widths of these types are not fixed, but usually `double` has 64 bits, `float` 32 bits, and `long double` 80 or 128 bits. As of C++23, there are also fixed-width types (see [Table 4.10](#)).

```
// https://godbolt.org/z/8ae83nPYs
#include <iostream>      // cout
#include <iomanip>        // setprecision, etc.
#include <cmath>         // fabs
using std::cout;        // cout as abbreviation for std::cout
int main() {
    cout << std::fixed << std::setprecision(25);    // for better readable output
    // 0.1 and 0.01 cannot store double exactly
    double x = 0.1 * 0.1;
```

```

cout << "0.1*0.1: " << x << "\n";
// Output: 0.1*0.1: 0.0100000000000000019428903
if(x == 0.01) {      // ✗ never compare double with ==
    cout << "Yes! x == 0.01" << "\n";
} else {
    cout << "Uh-oh! x != 0.01" << "\n";           // you see this output
}
// Attention especially when comparing with 0.0
double null = x - 0.01;
cout << "null: " << null << "\n";
// Output: null: 0.0000000000000000017347235
if(std::fabs(null) < 0.00000001) {                // compare with an "epsilon"
    cout << "Yes! null is close to 0.0" << "\n";   // you see this output
} else {
    cout << "Uh-oh! null not close to 0.0" << "\n";
}
// fractions of powers of 2 are less critical
double y = 0.5 * 0.5;
cout << "0.5*0.5: " << y << "\n";
// Output: 0.5*0.5: 0.250000000000000000000000000000
if(y == 0.25) {      // here the dangerous comparison works exceptionally
    cout << "Yes! y == 0.25" << "\n";           // you see this output
} else {
    cout << "Oh-oh! y != 0.25" << "\n";
}
//
return 0;
}

```

Listing 4.32 “double” cannot always store numbers exactly. Calculating and comparing with “==” is an error.

Do Not Compare Floating-Point Numbers with ==

Because none of the floating-point types, `float`, `double`, and `long double`, can store all numbers exactly, you should never test for equality by using `==`. Instead, calculate the difference and check whether the result is close to zero. For example, if you want to know whether the two `double` values `a` and `b` are “almost equal,” you can check this in this way:

```
if( fabs(b-a)<0.0001 )...
```

The `fabs()` function determines the absolute value (makes it positive) and is located in the `<cmath>` header. For 0.0001, use an *epsilon* that suits your application.

Floating-Point Literals

Similar to integers, you can write floating-point numbers naturally in the source text. However, you must use the dot `.` as a decimal separator; the continental comma does not work here. In the literal `124.258`, the compiler recognizes from the dot `.` that it is not an integer.

```
// https://godbolt.org/z/qx8h7a4j9
#include <iostream>
#include <iomanip>                                // fixed, setprecision
using std::cout;                                // abbreviated cout
int main() {
    cout << std::setprecision(2) << std::fixed;    // two decimal places
    cout << "1/4: " << 0.25 << "\n";              // comma notation for double
    // output: 1/4: 0.25

    cout << "2/4: " << 0.5 << "\n";
    // Output: 2/4: 0.50
    cout << "3/4: " << 0.75 << "\n";
    // Output: 3/4: 0.75
    cout << "4/4: " << 1 << " or " << 1.0 << "\n"; // ⚡ recognizes 1 as int
    // Output 4/4: 1 or 1.00
    cout << "1e0: " << 1e0 << "\n";                // scientific notation
    // output: 1e0: 1.00
    cout << "0x10.1p0: " << 0x10.1p0 << "\n";      // hexadecimal notation
    // output: 0x10.1p0: 16.06
}
```

Listing 4.33 Several ways to mark “double” literals.

If, as with `<< 1 <<`, the floating-point number happens to have no decimal places, then write at least the dot to distinguish it from the integer literal—and zeros if you want: `1.`, `1.0`, `1.00`, `1.000`, and so on. A `1` without a dot is an `int` for the compiler. This still works in many cases, because if the compiler knows that you actually mean a `double`, it converts the `int` `1` into a `double` `1` without loss. However, in cases where both would be possible, you should (or must) indicate the floating-point number. The output operator for `<< 1 <<` does not know whether you mean an `int`, so it outputs the `1` as such. The compiler recognizes the `1.0` as a floating-point number and the output is correct.

Since C++17, you can also use *hexadecimal notation for floating-point literals*, as you can see in the example at `0x10.1p0`. This is close to the internal representation, which we will discuss a little later in this section. In short, these are the building blocks of the literal:

- `0x`, the hexadecimal prefix
- `10.1`, the significand in hexadecimal notation (here, decimal `1.125`)

- `p`, the separator
- `0`, the exponent in hexadecimal notation

The significand or parts of it are optional, but `p` and the exponent always occur.

The type of such a floating-point literal is `double`. In most cases today, it makes sense to use this data type as the first choice when calculating with floating-point numbers. Today's hardware is designed for calculating with it. If you need to save space, want to use a third-party API, or have other reasons, make a floating-point literal a `float` by appending an `f` or `F` to it. Appending an `l` or `L` indicates a long double. As of C++23, there are extended floating-point types that have suffixes such as `f128` and the like (see [Table 4.9](#)).

```
// https://godbolt.org/z/6qfo4oYnr
#include <iostream>
#include <iomanip> // fixed, setprecision
int main() {
    using std::cout;
    cout << std::setprecision(30) << std::fixed;    // always output 30 digits
    cout << 1.111222333444555666777888999f << "\n"; // float literal
    // Output: 1.1112223863601684570312500000000
    cout << 1.111222333444555666777888999 << "\n"; // double is default
    // Output: 1.111222333444555676607023997349
    cout << 1.111222333444555666777888999d << "\n"; // double literal
    // Output: 1.111222333444555676607023997349
    cout << 1.111222333444555666777888999L << "\n"; // long double
    // Output: 1.111222333444555666740784227731
}
```

Listing 4.34 Floating-point literals become imprecise at some point.

There are many ways to output floating-point numbers or prepare them for output:

- `format("{", number)`
As of C++20, you can convert any arguments into a string. Format attributes for width, precision, and more are available for numbers. You can combine `cout << format("{", number)` as well.
- `to_chars(buffer, buffer + size, number)`
This function writes numbers to a buffer. There are optional arguments for formatting and an error check. You can use it from C++17 on.
- `to_string(number)`
This is only possible with numbers. You have no control over the format and can do little to check for errors.

- `cout << number`

The direct output of almost any type to a stream. For numbers, there are manipulators such as `setprecision` that influence the format.

- `print(cout, "{}", number)`

As of C++23, you can output any arguments to a stream as with `format`. The implementations are not yet complete as of the end of 2023.

The Floating-Point Features

Unlike integers, the limits and accuracy of floating-point types are the same on almost all systems. You would have to have a venerable old system if there were to be any differences here.

The pitfalls in dealing with floating-point numbers actually lurk elsewhere—namely, as mentioned earlier, in the realms of precision and rounding decimals. There are also operations in the “dark areas” of mathematics. For example, a floating-point variable can not only store a valid number, but can also take on a few very special values:

- If you divide by 0.0, this does not trigger an exception for floating-point numbers. The result is then *infinite*. Check a variable `x` with the function `std::isfinite(x)` from the header `<cmath>` to see whether it still represents a finite value.
- Only 0.0/0.0 results not in infinity, but in “not a number” (*NaN*), just like `sqrt(-1.0)`, the square root of -1. All further calculations with NaN result in NaN again. You can use the `std::isnan(x)` function to check a value for this.

You can learn more about the even more technical details in the next sections.

Floating-Point Internals

If you really want to understand how floating-point numbers work in C++, you have to deal with their internal representation in C++. This is based on the IEEE 754 standard. Even if you don't manipulate the bits of the floating-point number individually, it helps to understand what the computer does, *can* do, and especially *cannot* do.

The floating-point types available by default are `float`, `double`, and `long double`. For the sizes of these types, it is only specified that they are equal or ascending; the details depend on the compiler used and the target platform. In x86 in practice, a `float` has 32 bits, a `double` 64 bits, and a `long double` 80 bits or, more rarely, 128 bits. This corresponds to the IEEE 754 standard sizes, which you can find in [Table 4.9](#).

With the 64 bits of a `double`, you can store numbers as large as 10^{300} , and amounts as small as $1/10^{300}$ can be stored. The bits are divided internally into three groups: the *sign*, the *exponent*, and the *significand* (also called the *mantissa*). The number is then stored scaled, in binary, according to this formula: $V \times 1.S \times 2^E$.

Pitfalls: No Streams and No Aliases

You can format the new types with `std::format("{ }", val)`, but types wider than `long double` are not supported for streams.

For example, even if the type `float64_t` is a `double` on the 64-bit Linux, `float64_t` is not an alias of `double`, but a separate type. This means that you can have overloads for both types:

```
void f(double);
void f(float64_t); // okay: other overload
```

This is different from the `int` types, where `int32_t` is an alias for `int` and such overloads are acknowledged by the compiler with an error. Note, however, that you should normally choose *one* of the two variants for your code, either the classic types or the types with a defined layout.

An implementation—that is, a compiler and standard library—can offer further types, but these must then be divisible by 32. Whether a type is available can be checked with the `__STDCPP_Typ__` macro—for example, `__STDCPP_FLOAT128__`.

Name	suf	bits	d2/d10	max e2	max()	eps()	Time
<code>float16_t</code>	<code>f16</code>	16	11/3	16	6.55e+04	0.0009766	208ms
<code>bfloat16_t</code>	<code>bf16</code>	16	8/2	128	3.39e+38	0.007812	191ms
<code>float</code>	<code>f</code>	32	24/6	128	3.403e+38	1.192e-07	40ms
<code>float32_t</code>	<code>f32</code>	32	24/6	128	3.403e+38	1.192e-07	40ms
<code>double</code>		64	53/15	1024	1.798e+308	2.22e-16	40ms
<code>float64_t</code>	<code>f64</code>	64	53/15	1024	1.798e+308	2.22e-16	40ms
<code>long double</code>	<code>l</code>	128	64/18	16384	1.19e+4932	1.084e-19	81ms
<code>float128_t</code>	<code>f128</code>	128	113/33	16384	1.19e+4932	1.926e-34	514ms

Table 4.10 The floating-point types added in C++23 in comparison to the classic ones (in bold) on a 64-bit Linux system.

Let's go into even more detail. I will use an example to explain the exact conversion of a floating-point number into the machine representation of the `float`. When I use the term *floating-point number*, I mean the mathematically abstract notion of a number, but now specifically in the decimal system we are used to. By `float`, I mean the *IEEE 754* standardized binary representation of a floating-point number in *single precision*.

As an example, I take the number 45.45 and convert it into a 32-bit float representation with one sign bit, eight bits for the exponent, and 23 bits for the significand.

The sign is quickly converted. The + for a positive number is saved as a 0 bit. A negative number would result in a 1 bit. There’s not much there yet, but in the machine it now looks as shown in [Figure 4.5](#).

V	Exp. [8 Bits]	Significand (S) [23 Bits]
0		

Figure 4.5 The beginning of the conversion of a floating-point number.

The next step is to convert the decimal notation into binary notation. Here we take care of the parts before and after the comma separately.

You can treat the part before the decimal point—in this case, 45—as an integer. You divide this number by 2 and write down the remainder (0 or 1) until there is nothing left, as described in [Section 4.3](#) (see also [Listing 4.25](#)).

For $_{10}45$, you then get binary $_2101101$.

Proceed slightly differently for the decimal part, .45. Multiply the number repeatedly by 2 and note the 0 or 1 before the decimal point. Ignore the part before the decimal point for the next step. Continue until you have already carried out a multiplication operation once:

.45 × 2 = 0.9	
.9 × 2 = 1.8	
.8 × 2 = 1.6	<---+
.6 × 2 = 1.2	
.2 × 2 = 0.4	
.4 × 2 = 0.8	
.8 × 2 ... we already had	-+

You could continue endlessly, but it is more practical to note this period somehow—for example, with a line above the repeating part: $0.1\overline{1100}$. If the repeating part were a zero, the result would be exact and the number would have no period in binary notation.

The parts before the decimal point and after the decimal point now add up to the following in binary notation: $_2101101.01\overline{1100}$.

You can now determine the exponent from this representation. To do this, move the decimal point after the first 1 and count how many positions you have moved—five positions in this case. In *scientific notation* in the decimal system, you could now write “e5” after the scaled number. In this case, you first convert the integer 5 into a binary number and obtain $_210110101\overline{1100}e101$.

You now almost have the exponent for the machine representation. So that exponents smaller than zero can also be saved, you now only need to add 127 (for `float`) to the exponent—that is, 132 for 5, or binary 210000100 .

There are eight bits, just as much space as in the exponent for a `float`. If there were fewer, you would pad with zeros in front; if there were more, you would have an overflow and the number could only be stored as infinity. If the result for the exponent were negative, this would be an *underflow*—a number so close to zero that it can only be stored as zero.

Now I'll write the bit sequence as an exponent in the machine representation (see [Figure 4.6](#)).

V	Exp. [8 Bits]	Significand (S) [23 Bits]
0	10000100	

Figure 4.6 Here, 127 was added to save the exponent.

In the significand, the computer only stores the digits after the decimal point. This procedure guarantees that there is always exactly 1 before the decimal point. Only in the case of *subnormal numbers*, if the exponent to be stored were exactly zero, there would be no 1. The illustration deals with this case in particular.

Subnormal Numbers

If the exponent to be stored in bit representation is exactly zero, then the conversion of the significand is slightly different—"not normal." The number is then *denormalized* or *subnormal*. In this case, the computer assumes that there is no 1 before the decimal point in binary notation. If it becomes necessary to store a number in denormalized form, it's a sign that you're on the verge of an underflow. I won't delve further into that here.

It is typical that calculations on denormalized numbers are much slower. You can check whether a number `x` is denormalized by calling `std::fpclassify(x)` from `<cmath>`. If the result is `FP_SUBNORMAL`, then the variable `x` is close to an underflow.

As a final step, I transfer the decimal places from $21.01101011100\overline{e}101$ into the significand. I roll out the period so long as I have space in the 23 bits. Without a period, I would fill it with zeros. The result is shown in [Figure 4.7](#).

V	Exp. [8 Bits]	Significand (S) [23 Bits]
0	10000100	01101011100110011001100

Figure 4.7 The number 45.45 in IEEE 754 with single precision.

You cannot store a period in a finite bit sequence in this way. This is why some numbers that you can easily write down on paper in decimal notation do not have an exact representation for the computer. The preceding representation results in 45.449996 when recalculated. Some numbers can be stored exactly, such as 0.5, but even 0.1 gives you a period in binary notation.

Special Values

Some bit patterns have a special meaning and help the computer (and you) to detect and handle edge cases. If you have a floating-point number x , you can use the function `std::fpclassify(x)` from `<cmath>` to determine whether you have such a special number:

- If the exponent stores zero as a bit sequence, it is a *subnormal number* very close to zero and therefore just before an *underflow*. `fpclassify` returns `FP_SUBNORMAL` for such numbers.
- For numbers that are too large, such as $1.0/0.0$, there is *positive infinity*, $+\infty$. The function `fpclassify` is `FP_INFINITE`; `signbit(x)` returns 0.
- For numbers in the negative range such as $-1.0/0.0$, there is *negative infinity*, $-\infty$. `fpclassify` is `FP_INFINITE`; `signbit(x)` returns 1.
- Invalid numbers such as $0.0/0.0$ and `sqrt(-1.0)` are called *NaN*. `fpclassify` is `FP_NAN`.
- If a calculation is too close to zero in terms of precision, the computer saves $+0.0$ and -0.0 as separate values. `x==0.0` is true for both, and `fpclassify` returns `FP_ZERO` for both. If you need to distinguish them, use `signbit(x)`, which returns 1 for -0.0 .

```
// https://godbolt.org/z/Tvx4qcrE6
#include <iostream>
#include <cmath>    // fpclassify
#include <limits>   // numeric_limits
#include <string>

std::string fpclass(double x) {
    switch(std::fpclassify(x)) {
        case FP_INFINITE: return "infinite";
        case FP_NAN:      return "NaN";
        case FP_NORMAL:   return "normal";
        case FP_SUBNORMAL: return "subnormal";
        case FP_ZERO:     return "zero";
        default:          return "unknown";
    }
}
```

```

int main() {
    const auto dmin = std::numeric_limits<double>::min();
    std::cout
        <<"1.0/0.0 is "<<fpclass(1/0.0)<<'\n'    // Output: 1.0/0.0 is infinite
        <<"0.0/0.0 is "<<fpclass(0.0/0.0)<<'\n'    // Output: 0.0/0.0 is NaN
        <<"dmin/2 is "<<fpclass(dmin/2)<<'\n'      // Output: dmin/2 is subnormal
        <<"-0.0 is "<<fpclass(-0.0)<<'\n'          // Output: -0.0 is zero
        <<"1.0 is "<<fpclass(1.0)<<'\n';           // Output: 1.0 is normal
}

```

Listing 4.35 Among other things, you can discover special values with “fpclassify.”

The header defines macros or constants for the special values. `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` contain very large values, `INFINITY` the positive infinity, and `NAN` not a number.

In addition to `fpclassify`, other functions are available in `<cmath>` to determine the properties of floating-point numbers. `isfinite` (not `NAN` or `INFINITY`), `isinf` (`INFINITY`), `isnan` (`NAN`), `isnormal` (not `INFINITY`, `NAN`, zero, or subnormal), and `signbit` (`sign`) have the obvious meanings. `isunordered` checks whether at least one of two numbers is `NAN`.

Tips for Almost Exact Calculation

When adding, it is better to add large numbers to large numbers and small numbers to small numbers rather than large numbers to small numbers.

Multiplication is a little trickier. Due to the scaling in the exponent, calculations between -1.0 and $+1.0$ are more accurate than those further away from 0.0 . It is advisable to bring calculations as close to this range as possible. This means, if you can influence it, arrange the multiplications in such a way that you follow this rule: “Large to small, small to large.”

This means that $(0.1 * 10.0) * (10.0 * 0.1)$ is better than $(0.1 * 0.1) * (10.0 * 10.0)$, but mainly for extreme values. This means that the intermediate results do not move further away from 1 than necessary.

I think a detailed example will help you to see what can happen if the calculation elements are arranged unfavorably. For example, there is *Heron’s formula* to calculate the area of a triangle given three side lengths (a , b , c):

Rewritten slightly, Heron’s formula in C++ looks like this:

```

float heron(float a, float b, float c) {
    auto s = (a+b+c) / 2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

```


For comparison, there is the *Kahan method*,⁸ which first sorts the side lengths of the triangles according to $a \geq b \geq c$ and then looks for a better grouping.⁹ After sorting, the following formula results:

$$A = \frac{1}{4} \sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}$$

However, only if $c \leq a - b$, because otherwise the sides do not form a triangle. In C++, it looks like this:

```
float kahan(float a, float b, float c) {
    auto x = max(a, max(b, c));
    auto y = max(min(a, b), min(max(a, b), c));
    auto z = min(a, min(b, c));
    return sqrt( (x+(y+z))*(z-(x-y))*(z+(x-y))*(x+(y-z)) )/4;
}
```

For a right-angled triangle with the side lengths 3, 4, and 5, the area 6 can be calculated in the head using the Pythagorean theorem. The following program is the same for both methods. I have used the template variable `T` instead of `double` as the type in order to be able to execute the same code with `double` and `float` (see [Chapter 23](#)):

```
// https://godbolt.org/z/We3oWrcWW
#include <iostream>
#include <cmath> // sqrt
#include <concepts> // floating_point
using std::min; using std::max; using std::floating_point;

template<floating_point T> T heron(T a, T b, T c) {
    auto s = (a+b+c) / 2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

template<floating_point T> T kahan(T a, T b, T c) {
    auto x = max(a, max(b, c));
    auto y = max(min(a, b), min(max(a, b), c));
    auto z = min(a, min(b, c)); return
    sqrt( (x+(y+z))*(z-(x-y))*(z+(x-y))*(x+(y-z)) )/4 ;
}
```

⁸ *Miscalculating Area and Angles of a Needle-like Triangle*, paragraph 2, W. Kahan, <https://people.eecs.berkeley.edu/~wkahan/Triangle.pdf>, [2014-09-04]

⁹ *Demystifying Floating Point*, John Farrier, CppCon 2015

```
template<floating_point T> void triangle(T a, T b, T c) {
    std::cout << "heron: " << heron(a,b,c) << '\n';
    std::cout << "kahan: " << kahan(a,b,c) << '\n';
}

int main() {
    triangle(3.0f, 4.0f, 5.0f);
}
```

The output is then 6 for both procedures.

Funny things happen when you insert the side lengths (100000, 99999.99979, 0.00029) and let the computer calculate the area. I run both calculations for `float` and `double` and output more digits:

```
// https://godbolt.org/z/863b1z4GY
int main() {
    std::cout << std::setprecision(15) << std::fixed;
    triangle(100'000.0f, 99'999.999'79f, 0.000'29f);
    triangle(100'000.0, 99'999.999'79, 0.000'29);
}
```

The exact result would be 10. The example was carefully chosen so that `float` has difficulties in the calculation: while the classic Heron formula gives 0, the Kahan formula calculates 14.5. And 14.5 would look like a nice result if you didn't know that 10 is correct. Even though both methods give a better result with `double`—because I told you that 10 should come out—such an example could also have been found for `double`. The actual outputs are as follows:

```
heron: 0.000000000000000
kahan: 14.500000000000000
heron: 9.999999809638329
kahan: 10.000000077021038
```

However, it is even more important that you have to perform as *few* calculations as possible. Let's assume you work in video editing and want to calculate the length of a movie. Videos often have one frame per 1/25 of a second. Then you can calculate the current movie time in a loop in two ways. First, try it accumulated, and second, try it with closed multiplication, as the following two listings demonstrate.

```
// https://godbolt.org/z/sEE94bh5K
#include <iostream>
#include <iomanip> // setprecision, fixed
constexpr int framesPerSec = 25;
constexpr int runtimeInSecs = 3600;
```

```
constexpr int framesTotal = runtimeInSecs * framesPerSec;
constexpr float frametime = 1.0f / framesPerSec;

int main() {
    float filmtime = 0.f;
    for(int n=1; n <= framesTotal; ++n) { //1..framesTotal
        filmtime += frametime;           // accumulate
        // ... here code for this frame ...
    }
    std::cout << std::setprecision(10) << std::fixed
        << filmtime << '\n';           // output: 3602.2695312500
}
```

Listing 4.36 Accumulated time measurement.

You shouldn't be surprised that more than two seconds of errors have accumulated in one hour of film.

Labeling Constants

For `constexpr`, refer to [Chapter 13, Section 13.10.6](#). It is advantageous for the compiler and for you if you mark unchangeable values (i.e., constants) as such. You can use (static) `const` or `constexpr` for this purpose. The latter ensures that the compiler can calculate the constant in advance. If it cannot do this, it will inform you accordingly.

I let the loop run from 1 to `<= framesTotal` to be able to compare it better with the next example.

However, you can also approach this differently. So long as the `frametime` remains constant over the course of the movie, you can calculate it in one step. So you are scaling as in the following listing.

```
// https://godbolt.org/z/dKP07andE
#include <iostream>
#include <iomanip> // setprecision, fixed
constexpr int framesPerSec = 25;
constexpr int runtimeInSecs = 3600;
constexpr int framesTotal = runtimeInSecs * framesPerSec;
constexpr float frametime = 1.0f / framesPerSec;
int main() {
    float filmtime = 0.f;
    for(int n=1; n <= framesTotal; ++n) { //1..framesTotal, because of formula
        filmtime = frametime * n;         // scale
        // ... here code for this frame ...
    }
}
```

```
std::cout << std::setprecision(10) << std::fixed
    << filmtime << '\n';           // output: 3600.0000000000
}
```

Listing 4.37 Closed time calculation.

And there is no longer any deviation.

Here you can see why I preferred to run from 1 to `<= framesTotal` in the `for` loop instead of from 0 to `< framesTotal`: only then does `filmtime = frametime * n` actually correspond to `filmtime = frametime * framesTotal` in the last loop pass.

Tips on the Calculation Sequence

- When adding, keep the large numbers together with other large numbers and the small numbers together with the small ones.
- Carry out multiplications as close to the interval from -1.0 to +1.0 as possible.
- Perform as few calculations as possible so that errors are less likely to propagate. For example, prefer a single multiplication to a repeated addition.

These rules apply if you have decided to calculate with floating-point numbers. Before doing so, you should ask yourself whether integers are not better suited to your task. Integers are fast and precise with the right operations. However, they do not cover such a large calculation range.

When calculating *amounts of money*, you should *never* (never! never! never!) calculate with floating-point numbers. Calculate in hundredths of a cent or similar and do not continue to calculate with the rounded values that you need for the output.

From Zero and Less Than

Because indexes usually start at zero, it is common in C++ to also start loops at zero. This means that you normally run `for` loops up to `< size` when they run through a C-array or vector, for example. If the loops exceptionally start at 1 or run up to `<= size`, you should mark this clearly. Otherwise, speedy readers might overlook it out of habit or wonder whether you have made a careless mistake.

4.4.5 Truth Values

The data type with which you can represent true or false is `bool`. You can declare `bool` variables and cache truth values in them. Many functions return a `bool` or take one as an argument. Just as there are literals for integers, floating-point numbers, and so on, there are also literals for `bool`: namely, `true` and `false`.

This data type is also essential in many C++ language elements, as conditions are always written as `bool` expressions. This means that an expression for the condition in an `if` or `for` or the like is converted into a `bool`. For example, all comparisons have `bool` as the result, like `<`, `<=`, `>`, and `>=` as well as `==` and `!=`:

```
if(a >= b) ...
for(int idx; idx < 100; ++idx) ...
while(!finished && linesRead < LIMIT) ...
```

You can link two `bool` expressions with each other. Write “Are *a* and *b* both true?” as `a && b`. Find out whether at least one of them is true with `a || b`. Turn true in *a* into false and vice versa with `!a`. Because there are only two possible values for `bool`, [Table 4.11](#) contains all possible combinations. The somewhat unusual exclusive OR operator `^` is also included there. And last but not least, you can of course put expressions in parentheses with `(...)`. You can find out more about these operators and expressions in [Section 4.3](#).

a	b	Operator	Expression	Result
false	false	&&	a && b	false
false	true			false
true	false			false
true	true			true
false	false		a b	false
false	true			true
true	false			true
true	true			true
false	false	^	a ^ b	false
false	true			true
true	false			true
true	true			false
false		!	!a	true
true				false

Table 4.11 Truth table for all Boolean expressions with two values.

It would actually be sufficient for the computer to output a bit to store a `bool`. However, the computer actually uses at least one `char` and more often an `int` for this. This is because the computer always blocks an entire “memory word” when manipulating bits and would therefore slow down neighboring bits. So you spend a little more space on independence and speed.

If you want to save a large number of individual `bool`s, you should take a look at the `std::bitset` data type in the standard library.

This relationship with `int` is one of the reasons that you can also simply use an `int` in tests with `if` and the like in C++:

```
int idx = 100;
while(idx) { // an int as a test
    readLine(line);
    --idx;
}
```

This loop reads in exactly 100 lines. As soon as `idx` reaches zero, the loop is terminated. The `int` value of `idx` is *implicitly* converted into a `bool`. This is done according to the following rules, which take some getting used to:

- A value of 0 is converted to `false`.
- All other values become `true`.

Because many C++ programmers originally come from C, where there was no pure `bool` data type, you will encounter this pattern quite often.

You can also use other data types in tests. To do this, the data type only needs to have the operator `bool()` method. This is the case, for example, with `std::istream` and therefore with `std::cin`. You can see an application example in the `while` loop in [Chapter 27, Listing 27.6](#).

4.4.6 Character Types

What is a “character” in abstract terms? It is a building block of a text that is to be displayed or processed. In C++, many things are mixed up here, and some things are not intuitive.

The main data type for characters is `char`. At the same time, however, you can also regard `char` as an integer data type, as you can calculate with it in the same way as with all other `int` relatives. And yet it is different here, because it is not specified whether `char` is signed or unsigned without further specifications. So if you want to calculate with a `char` and the range is important, then you should always write `signed char` or `unsigned char` instead of simply `char`; reserve the latter for texts.

“char” Is the Smallest “int”

It is even specified that `char` is the smallest integer data type and something like a base unit for memory usage. If you apply `sizeof(...)` to any type or value, you get its size in `char` units:

```
sizeof(char);    // 1
char x;
sizeof(x);       // 1
int value;
sizeof(value);   // 4 or something else on other systems
```

On current systems, you can assume that a `char` has eight bits. Because you cannot rely on the sign, you should only assume that the number range from 0 to 127 is available to you; calculating outside this range can lead to overflow (at the positive or negative end).

Now let's move on from number interpretation to text interpretation. If you want to combine `char` elements into text, then you normally do this as follows:

- In `std::string` if you want to save, pass around, or manipulate the text
- In `const char[]` if you are outputting a literal or using an old C API

A text literal such as "Hello" is made up of individual `char` elements. The first is an 'H', the second an 'a', and so on. Note the single quotation marks '...' around the respective character: these make the character a `char`. You cannot combine several characters in such a `char` literal; for example, 'botch' makes no sense.

If you initialize a `char` variable with a literal or assign one to it, then you are free to do this with quotation marks '...' or numbers, depending on how it fits better in the context:

```
char lower_a = 'a';           // as a character
char letters_in_the_alphabet = 'z'-'a'+1; // do the math
char small = 5;               // as a number
signed char less = -10;       // negative only with explicit signed
unsigned char large = 200;     // large only with explicit unsigned
char flags = 0x7f;           // hexadecimal for bit pattern 01111111
```

International Signs

In addition to `char`, there are other character types. As `char` is only suitable for 256 different characters, it would not be suitable for storing Chinese characters, for example. There are character types with more space for this:

- `char32_t`

With its 32 bits, this is sufficient for the most obscure international characters. Every possible Unicode character can be represented with a single `char32_t`. With the

prefix `U`, you write a literal such as `char32_t zed = U'Z'`; and character strings in `std::u32string`.

■ `char16_t`

This character type, which is always 16 bits wide, is good for 65,536 different characters and is sufficient for most international texts. Use the prefix `u` before a character literal—for example, `char16_t zed = u'Z'`; . The character string type `std::u16string` is based on this. A single `char16_t` is sufficient for the vast majority of Unicode characters. Only very rarely do you need two consecutive ones, which are then encoded in UTF-16.

■ `char8_t`

This type is added in C++20. It accepts UTF-8 code units and is therefore at least eight bits wide, just like `char` and `unsigned char`, but is different as a type. Only ASCII characters fit into a single `char8_t`. This means that even for a Unicode *ä*, you need two `char8_t`—that is, `0xC3` followed by `0xA4` in UTF-8 encoding. You write literals of this type with a `u8` prefix—that is, `u8'a'` for characters and `u8"gr\uF6n"` for character strings.

■ `wchar_t`

Depending on the system, this character type has 16 bits (Windows) or 32 bits (many Linux variants). If you write an `L` in front of a character literal, it is of this type—for example, `wchar_t zed = L'Z'`; . As it can be of different widths, you should not use this type in programs that have to run on several systems. The character string type `std::wstring` is based on this type; to output it, use `std::wout` or `std::werr` or as input `std::win`.

There are many ways to save a `Z`. To make the differences clearer to you, [Table 4.12](#) shows the representations for the computer.

Character Type	Literal	Interpretation in Bits
<code>char</code>	<code>'Z'</code>	0101'1010
<code>char8_t</code>	<code>u8'Z'</code>	0101'1010
<code>wchar_t</code>	<code>L'Z'</code>	0000'0000'0101'1010
or		0000'0000'0000'0000'0000'0000'0101'1010
<code>char16_t</code>	<code>u'Z'</code>	0000'0000'0101'1010
<code>char32_t</code>	<code>U'Z'</code>	0000'0000'0000'0000'0000'0000'0101'1010

Table 4.12 The different ways to write a `Z` with the different character types.

With a `Z`, this is not very exciting. But you can't write a Θ (theta) in the source code. You have to use an escape sequence for this. The Unicode character theta has the number ₁₆0398, so you can write `u'\u0398'` in a `char16_t` character literal or `U'\U0000398'` in a

`char32_t` character literal. The `\u` notation requires a character number with exactly four hexadecimal digits after `\u` or exactly eight after `\U`. Four digits should usually be sufficient. Only for something as exotic as the Gothic letter *ahsa* would you have to write `U'\U00010330'`.



Figure 4.8 The Gothic *ahsa* character with the hexadecimal Unicode number 10330 must be escaped with “U.”

The correct character should then appear on the correct medium—for example, in dialog boxes or file streams that can accept characters or strings of these types.

Unicode in C++

Internationalization, Unicode, and encoding are very complex topics that we cannot go into further in this book. It is best to limit yourself to `char` first and then read up on the use of the other character types if you need to.

With the introduction of `char8_t` and `u8string` in C++20, we have come a big step closer to full Unicode support. The standardization of encodings will still take some time, but it is planned.

4.4.7 Complex Numbers

If you know what *complex numbers* are, you may also want to calculate with them. This is possible because the `complex` template data type from the `<complex>` header is part of the standard library.

For those who do not know what this means, here is a brief explanation. You probably know that *whole numbers* are the numbers 0, +1, -1, +2, -2, +3, and so on. You can do anything in this number range until you discover multiplication and the division that goes with it. Then you need the *rational numbers*; these are any two whole numbers in a fraction—so, for example, $-3/4$. You are happy with this number range until you get the idea of adding very long fractions that get smaller and smaller. This is practical if you want to give all points on a line a name. The sum of such sequences results in *real numbers*, which include π and $\sqrt{2}$, for example. Do we have everything now? Yes—so long as you don't want to calculate the square root of a negative number. Because every real number squared results in a positive number (or zero)—for example, $-4^2 = 16$ —it is not easy to define what the square root of a negative number should be. For example, what is $\sqrt{-1}$?

Such a gap robs mathematicians of sleep, which is why they define it as follows: let $\sqrt{-1}=i$ or $i^2 = -1$. Phew, that helps mathematicians: now they can solve equations that have no solution in real number space. For example, $(x + 1)^2 = -9$ does not result in a true statement for any real number x . However, if we take the earlier definition, we see that x has the two solutions: $-1 + 3i$ and $-1 - 3i$. It looks strange, but you get used to it. A complex number such as $-1 + 3i$ always has two components: the *real* part without i , here -1 , and the *imaginary* part, here $+3i$. If you look at the two parts as x and y coordinates on a plane, you can see the complex numbers as points in the plane—just as you can see the real numbers as points on a line.

Special rules apply when calculating with complex numbers. They're not really anything special; you just have to keep in mind that every i^2 can be replaced with -1 . Otherwise, the normal arithmetic rules apply:

- **Addition:** $(3 + 5i) + (8 - 1i) = (3 + 8) + (5 - 1)i = (11 + 4i)$
- **Multiplication:** $(-2 + 3i) \cdot (4 + 2i) = -2 \cdot 4 + 3 \cdot 4i - 2 \cdot 2i + 2 \cdot 3 \cdot i \cdot i = -8 + 12i - 4i + 6i^2 = -8 + 8i - 6 = -14 + 8i$
- **Exponent:** $((-1 + 3i) + 1)^2 = (3i)^2 = (3^2) \cdot (i^2) = 9 \cdot (-1) = -9$

No sooner had this number space been discovered than physicists and mathematicians found all kinds of “practical” applications for it. In fact, complex numbers can be used to solve problems that were previously unsolvable.

Complex Numbers in C++

What does this look like in C++? If you use the `<complex>` header, the following things are available to you:

- **The template data type `complex<>`**
You can choose whether the imaginary and real parts should be calculated in `float`, `double`, or `long double` by specifying this as a template argument.
- **Operator overloads**
You can perform arithmetic calculations with `+`, `*`, and the like, and stream operations with `<<` and `>>` on `complex` variables.
- **Free auxiliary functions**
You get auxiliary functions for manipulation and calculation, such as `sin`, `pow`, and so on, as well as conversions, such as polar coordinates with `polar`.
- **User-defined literals**
Among other things, `operator""i` is defined in `std::literals::complex_literals` so that you can easily write complex literals in the source code.

If you output a complex number with `<<`, the number appears in the form `(real, imag)`.

```
// https://godbolt.org/z/43rYYeKx9
#include <iostream>
#include <iomanip> // setprecision, fixed
#include <complex>
using std::cout; using std::complex;

int main() {
    using namespace std::complex_literals; // for i-suffix
    cout << std::fixed << std::setprecision(1);
    complex<double> z1 = 1i * 1i;           // i times i
    cout << z1 << '\n';                     // Output: (-1.0,0.0)
    complex<double> z2 = std::pow(1i, 2);   // i-square
    cout << z2 << '\n';                     // Output: (-1.0,0.0)
    double PI = std::acos(-1);              // Length of half a unit circle
    complex<double> z3 = std::exp(1i * PI); // Euler formula
    cout << z3 << '\n';                     // Output: (-1.0,0.0)
    complex<double> a(3, 4);                // usually as a constructor
    complex<double> b = 1. - 2i;            // practically as a literal

    // Calculations:
    cout << "a + b = " << a + b << "\n";    // Output: a + b = (4.0,2.0)
    cout << "a * b = " << a * b << "\n";    // Output: a * b = (11.0,-2.0)
    cout << "a / b = " << a / b << "\n";    // Output: a / b = (-1.0,2.0)
    cout << "|a| = " << abs(a) << "\n";     // Output: |a| = 5.0
    cout << "conj(a) = " << conj(a) << "\n"; // Output: conj(a) = (3.0,-4.0)
    cout << "norm(a) = " << norm(a) << "\n"; // Output: norm(a) = 25.0
    cout << "abs(a) = " << abs(a) << "\n";  // Output: abs(a) = 5.0
    cout << "exp(a) = " << exp(a) << "\n";  // Output: exp(a) = (-13.1,-15.2)
}
```

Listing 4.38 You can calculate arithmetically with complex numbers.

Table 4.13 and Table 4.14 show the list of free functions for complex numbers.

Function	Description
real	Real part of a complex number
imag	Imaginary part
abs	Absolute value
arg	Angle, phase angle
standard	Standard, length

Table 4.13 Functions for accessing and converting complex numbers.

Function	Description
conj	Complex conjugation on the real axis
polar	Construction of polar coordinates
proj	Complex projection

Table 4.13 Functions for accessing and converting complex numbers. (Cont.)

With GCC, you have to use `-std=c++14`, not `-std=gnu++14`. The GCC extensions get confused here for `complex`; you have to switch to standard.

Functions	Description
sin, cos, tan, sinh, cosh, tanh	Trigonometric functions
asin, acos, atan, asinh, acosh, atanh	Inverse trigonometric functions
sqrt, pow, exp, log, log10	Square root, exponent, logarithms

Table 4.14 Transcendental functions for complex numbers.

4.5 Undefined Behavior

C++ defines that statements happen one after the other, but within a statement the standard leaves open the order in which things happen. For example:

```
std::cout << add(3+4);
```

It is clear that `3+4` is calculated before `add` is fed the result. Similarly, the output with `<<` can only occur after `add` has been returned. This is different in the output function in the following listing.

```
// https://godbolt.org/z/eW1cfYcGs
#include <iostream>
void output(int a, int b) {
    std::cout << a << ' ' << b << '\n';
}
int number() {
    static int val = 0;
    return ++val;
}
int main() {
    output(number(), number()); // in which order?
}
```

Listing 4.39 Output in unspecified order.

Here, the standard deliberately leaves open the order in which the two `number()` calls are made. This means that you can receive the output 2 1, but you can just as easily see 1 2. Both are okay. This freedom is called *unspecified behavior*. This is different from *undefined behavior*, which means something like “anything can happen here.” The program can continue running with incorrect values or it may crash. Both serious scenarios are mentioned multiple times in the standard.

Navigating the realms of unspecified behavior typically keeps you in the clear. However, behavior may differ on a different system, with a different compiler, or even upon revisiting the same point. The order in which function call parameters are evaluated is one such case.

A subcategory of this is *implementation-defined behavior*. Here, the standard leaves open how the compiler has to implement a feature; it must choose a type and document it. For example, an `int` can occupy 16 bits on some systems and 32 bits on others, but the manufacturer must document his choice.

However, if you have undefined behavior, then you usually have a bug. Even if the probability of your walls suddenly changing color is very low, your program crashing or performing an incorrect calculation is likely and undesirable. If you use memory that no longer belongs to you or write expressions such as `i++ + ++i`, then the result is undefined.

In contrast to a language like Java, C++ has deliberately left some things unspecified or even undefined in order to give compiler and hardware manufacturers more options for increasing performance.

Chapter 5

Good Code, 1st Dan: Writing Readable Code

Now you have learned a lot about the syntax of C++ and what a program *can* look like. But what *should* it look like? At the end of the day, it doesn't matter how you format your code, but there are advantages to sticking to certain conventions to make it easier to understand the source code. This is important, of course, because the source code itself is a central means of communication between the programmers involved. And now for the surprise: it is probably *you* that will read your own program the most. Experience has shown that it's "helpful" (an understatement) to understand your own code.

To make this easier, you should primarily adopt conventions that you adhere to yourself. Consistent style is the most important for quick understanding. This enables something akin to speed-reading, where you immediately grasp recurring patterns with a single glance and often don't need to look at the details. It's almost irrelevant which conventions you adopt, so long as you stick to them. Just use your common sense and ask yourself, "Will I still understand this in five years?"

Only as a second step should you look for "common" conventions or adhere to what helps your teammates. No, I don't mean that you should selfishly use your own style. But when it comes to a specific point in the program text where you have to choose between A or B, first ask yourself which you prefer—and if you don't have a good answer, then ask your team members. Because after you've asked the team the same question three times, you'll know the answer. And then common sense is enough.

So, adapt the style of your team and make it your own. With that, you'll be well-prepared, and colleagues will also look to you for style guidance.

After these general tips, I'll now give you some specific advice on which code conventions you can choose from.

5.1 Comments

Comment a lot, but do not interrupt the flow of reading. Comment *in addition* to the code; do not simply repeat what is already there. Instead of `i = i+1; // increase i`, write something like `i = i+1; // Attention: overflow possible`. I find it beneficial to extensively explain the intention before a function or section without delving into specific

code elements themselves. Then, between the lines of code, I strive to write only single-line comments.

The *C++ Core Guidelines* have a few useful rules about this. I will summarize them briefly because comments are always a controversial topic. In [Chapter 30](#), I go into more detail about such rules (with the identifiers from the core guidelines):

- NL.1: Don't say in comments what can be clearly stated in code.
- NL.2: State intent in comments.
- NL.3: Keep comments crisp.

5.2 Documentation

Consider automatically generating documentation from comments. This makes it easier to keep code and documentation synchronized. There are special tools for this, such as *Doxygen* (<http://www.doxygen.org>), among many others. Such generated comments then have a special format and only *supplement* your other comments; they do not *replace* them:

```
/** Calculate Fibonacci number.
 * @param n -- input number, must be greater or equal to 0
 * @return the n-th fibonacci number */
// Mathematically: fib(n) = { 0: 0, 1:1, else: fib(n-1)+fib(n-2) }
// Recursive and therefore slow implementation.
// To speed things up, the recursion can be replaced by a table with a loop.
// From an 'n' of about 50, 'unsigned' could overflow.
unsigned fib(unsigned n) {
    // Test for ==0 is sufficient, because argument is unsigned.
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    // Only works with n>= 2. Otherwise dangerous underflow possible.
    return fib(n-1) + fib(n-2);
}
```

The comment `/** Calculate... */` is meant for API documentation that can be given to the customer—the one who will use the function. Here, similar to what is usual in Java, I have used special `@` tags for formatting so that Doxygen understands it.

The rest of the comments are intended for readers of the program text, in case errors need to be searched for or extensions need to be added. First, `// Mathematically...` explains what the whole function is about, without going into the specific implementation. This is followed by general notes on the implementation at `// Recursive...`, which aids understanding, or on special features to look out for.

In the program text itself, there are then only a few single-line comments, such as `// Test for...` and `// Only works with...`, which supplement the source code.

Commenting well is a task that should not be underestimated. Comments should *complement* the source code, not try to *replace* it. It is said redundant comments like `cout << "bell"; // write bell` are worse than none at all. As an analogy, you can think of good comments like a cooking recipe: the program's instructions are "Take 500 grams of flour, stir," and so on, which even Martians can follow. With the comment "Bake a pizza Margherita," readers will only understand what you are doing if they have the background or are familiar with the program.

Of course, it would be best if your code was written so clearly that you didn't need any comments—that is, that your code was self-explanatory. This starts with the naming of functions, parameters, variables, and classes and extends to the structure of the functions, modules, and entire program. Don't be put off by the subjunctive mood: try to achieve this goal every day! Where this is not possible, use carefully measured comments as a last resort.

Many people prefer `//` to `/*...*/` comments, as this makes it easy to comment out an entire section of code. In practice, this means that if you only use `//` comments, you can easily deactivate a large piece of code with `/*` at the beginning and `*/` at the end. However, if `*/` is included, this commenting out does not work. Commenting out is often practical, at least within functions.

5.3 Indentations and Line Length

Get used to a uniform style with regard to the indentation depth. A consistent style increases the reading flow. Here, too, you should consult with your team members.

These are the usual conventions:

- Indentations should only be made with spaces or one tabulator.
- If using spaces, two or four spaces are common; rarely, eight are used.

My personal favorite is four spaces for indentation, but often you see two.

If you use the tabulator for indentation, your IDE certainly has a setting that allows you to set the display to convenient two-, four-, or eight-space counterparts.

Just don't do one thing: never mix space indentation with tab indentation—neither within a line nor within a file. If you do, a colleague with the wrong tab setting will not be able to read the code at all. If you visit a file that is indented with tabs, also indent your changes with tabs and vice versa. Alternatively, convert the indentations of the entire file before your change (and save this step as a separate change in the version management—e.g., Git).

Many programmers appreciate it when the lines do not run out of the editor to the right and want a manual line break after 72 characters. Many tools work with such a width. In higher-level languages than pure C, however, this 72-character limit is reached more quickly, which is why some teams have agreed on 80, 90, 100, 120, or even 160 characters. Ultimately, most modern tools can scroll horizontally. However, you should try not to exhaust this limit all the time: sensible breaks are best.

Where paper printouts still have to be possible in principle—for certifications and the like—the agreed-upon limit must be adhered to. It is better to create a nice wrap by hand before leaving it to the printer.

It is common to program with a fixed-width font—where `iiii` is as wide as `www`. I like to use proportional fonts myself, and do well with them, but I know that I am a very rare species.

5.4 Lines per Function and File

If you don't let your program slide into chaos, it will be easier to maintain and service in the long term. Think about where useful “interfaces” between the program parts are at the beginning or when making extensions, and don't burden your readers with this later.

Limit the number of lines per function. If a function grows beyond this, split it up at that moment and turn it into calls to subfunctions. A common convention is to set a limit of between 20 and 100 lines. My personal rule of thumb is that a function should still fit on the screen.

The same applies to program files (mainly *.cpp files). If a file grows too much, divide it into sensible subparts. However, do not split it up arbitrarily. If you are working with classes in an object-oriented way, a common convention is to have exactly one header and one implementation file per class. If the latter is several thousand lines long, consider whether you should perhaps divide up the object hierarchy differently.

5.5 Brackets and Spaces

Whether you place the curly brackets on the line with `if` and `for` or on the next line and then indent the brackets again or not is another question of taste and agreement within the team. One possible way to do it is this:

- Opening parenthesis to the line with the keyword
- Indent bracketed block
- Closing bracket back to indentation depth with the keyword
- Also surround single statements with brackets

For round parentheses (. . .) there are similar differences: Do you use a space after the `if` or `for` or the parenthesis directly after it? Are parentheses always surrounded by spaces, even after opening and before closing? A possible convention would be as follows:

- Space neither before nor after the opening parenthesis
- Space after the closing parenthesis, but not before
- A space after the comma in lists, but not before

This is roughly the style you will find throughout this book. Whether you adopt it or choose a different option is ultimately irrelevant, so long as you stick to one style. For comparison, [Listing 5.1](#) shows the style of this book and [Listing 5.2](#) shows another one.

```
// https://godbolt.org/z/eP77v498E
#include <iostream>
int func(int arg1, int arg2) {
    if(arg1 > arg2) {
        return arg1-arg2;
    } else {
        return arg2-arg1;
    }
}
int main(int argc, const char* argv[]) {
    for(int x=0; x<10; ++x) {
        for(int y=0; y<10; ++y) {
            std::cout << func(x,y) << " ";
        }
        std::cout << "\n";
    }
}
```

Listing 5.1 Indentation four, curly brackets in line with keyword, few spaces.

```
// https://godbolt.org/z/4TMn37qhE
#include <iostream>
int func ( int arg1, int arg2 )
{
    if (arg1 > arg2)
        return arg1 - arg2;
    else
        return arg2 - arg1;
}
```

```
int main ( int argc, const char *argv[] )
{
    for ( int x = 0 ; x < 10 ; ++x )
    {
        for ( int y = 0 ; y < 10 ; ++y )
            std::cout << func ( x, y ) << " ";
        std::cout << "\n";
    }
}
```

Listing 5.2 Indentation two, curly brackets only if necessary and in own line, more spaces.

5.6 Names

Each identifier can be written in a specific style with the following properties:

- Capital or lowercase character at the beginning
- Capital letters throughout
- Underline `_` or CamelCase for word boundaries
- Start or end with an underscore `_`

It makes sense to differentiate among different groups by these visual aspects and then always maintain this. The same applies here: get used to a style and discuss it with your colleagues.

One possibility is as follows:

- *Variables* start with a lowercase character and continue in *CamelCase*, as in `myValue`, `btnCancelNow`.
- *Class variables* use this style as well, but end with an underscore, for example, `data_`, `fileWrite_`.
- *Functions* and *methods* are formatted like variables, as in `getData()`, `assertEqual()`—or instead of CamelCase, they can use underscores, as in `get_data()`, `assert_equal()` (this is sometimes called *snake_case*).
- *Simple types* are set in lowercase with underscore, but with a suffix, as in `money_type`.
- *Classes* start with an uppercase character in CamelCase, as in `Data`, `VideoImage`.
- *Macros* are written everywhere in capital letters with underscores.
- *Constants* and enum elements should *not be* written in capital letters with underscores so as not to get in the way of macros.

Be a Chameleon

Perhaps it is not your job to write a program from scratch. If you are only making local changes to an existing file in a larger project, then adapt, take a step back. The source code remains most readable if you *do not* apply your own style to the code fragment you are touching but instead make your additions look exactly like the code around them.

Look at the indentations to see whether tabs or spaces have been used. Where were the `{}` placed, where were the lines wrapped, and how were the variables named? Only major blunders should you not adopt (e.g., there is no indentation at all, or all variables are named `i`).

Nothing is worse for overall readability than a mishmash of different styles. *Therefore:* Do not impose your style on your changes. Only beyond a certain threshold is it better to take the trouble of reformatting the entire file according to your own rules.

Chapter 6

Higher Data Types

Chapter Telegram

- **string**
Basic string type in C++.
- **wstring, u16string, u32string, u8string**
String types for international character strings.
- **ostream and ofstream**
General output data stream and the output data stream for a file.
- **istream and ifstream**
General input data stream and the input data stream for a file.
- **cout, cin, cerr and clog**
Predefined standard data streams for input and output.
- **operator<< and operator>>**
Operators for input and output to and from data streams.
- **Manipulator**
Construct to change the input and output format of a stream.
- **Container**
A container is an abstract data type that serves as a receptacle for data. The stored elements all have the same type, which you specify when creating the container. The interfaces of the various containers are very similar.
- **Sequence container**
Above all, `vector` is an all-around container and allows access by integer index.
- **Algorithm**
In C++, this usually refers to a function that works on elements in containers. Typically, algorithms are general and support many container types.
- **Pointer**
A type of reference to values that can assume the “no reference” state.
- **C-array**
A special form of container that works well with C, usually represented by a pointer and hopefully the end or length of the array.

First of all, there are two groups of data types that are not built into the language itself but are part of the standard library. You therefore need one or more `#include` in your source code.

We start with strings and streams:

- **Character strings**

In `std::string`, you store sequences of characters that you can manipulate, read in, and output.

- **Streams**

You are already familiar with the standard input and output streams `std::cin` and `std::cout`. These have the types `std::istream` and `std::ostream`. You can also use related types for the input and output of files—and for much more.

6.1 The String Type “string”

Strings are best (or at least normally) stored and manipulated in a `std::string`. This type is located in the `<string>` header of the standard library. Make sure that you add this header before you use `string`.

“string” and “char”

In C and many C++ programs, character strings are stored in `char*` or `char[]` and their `const` variants. This is generally fine, but here I will first discuss the much more C++-like variant `std::string`, which you should prefer to `char*` and `char[]` in many cases.

Let's change [Chapter 4's Listing 4.28](#) a little by adding the handling of strings in the following listing.

```
// https://godbolt.org/z/cv1qfcEd4
#include <iostream>           // cin, cout for input and output
#include <string>              // You need this header of the standard library

void input(
    std::string &name,      // as parameter
    unsigned &birthYear)
{
    /* Input still without good error handling... */
    std::cout << "Name: ";
    std::getline(std::cin, name); // getline reads a string
    if(name.length() == 0) {      // length is a method of string
        std::cout << "You have entered an empty name.\n";
        exit(1);
    }
}
```

```

    std::cout << "Birth year: ";
    std::cin >> birthYear;
}

int main() {
    /* data */
    std::string name;           // defines and initializes a string variable
    unsigned birthYear = 0;
    /* input */
    input(name, birthYear);
    /* calculations */
    // ...
}

```

Listing 6.1 Some possible uses of strings.

You can do all kinds of things with variables of type `string`. You can use functions that take a string as a parameter, such as `std::getline` with `std::getline(std::cin, name)`. In addition, each `string` variable has *methods* that you call with the dot `.`, such as `name.length()` in the if. Methods are special functions that always work on one of the variables with which they were called using dot notation.

6.1.1 Initialization

You can see at the beginning of `main()` how to define a `string`. Because it is not a built-in type, it is also initialized even though `no =` or similar has been specified. In this case, the following variants are equivalent:

```

std::string name;
std::string name{};
std::string name{""};
std::string name("");
std::string name = "";

```

Here, however, only the first two variants are completely identical. As a complex data type, `string` has a *constructor*—a special function that serves the specific purpose of initializing a new variable. If you do not specify an initialization, the constructor with no parameters is used. Since C++11, you can also construct with `{...}` so that `{}` makes the call of the constructor without parameters explicit. You can find out more about this in [Chapter 12](#).

In the case of `string`, a constructor with no parameters initializes the variable with the empty string—that is, `""`. Therefore, calling the constructor with the empty string as parameter `std::string name{""}` effectively does the same thing.

During initialization, the equal sign `=` does not mean “assignment”; instead, the constructor is called with a parameter. `std::string name = ""` therefore has the same effect as `std::string name{""}`.

Before C++11, the curly brackets were not available for initializing strings. The notation in the older standard was `std::string name("")`.

Attention When Initializing with Round Parenthesis without Parameters

You *cannot* use `std::string name()`; for initialization without parameters. In this case, *always use* the empty curly brackets—that is, `std::string name{};`, or before C++11, without brackets as `std::string name;`.

The empty parentheses `()` unfortunately have a double meaning here as an empty type list for a function declaration. The preceding function declares a function `name` without parameters with the return type `string`. Unfortunately, the compiler initially accepts this, but complains when `name` is used.

If you have C++11 or higher, you should get into the habit of always initializing variables with the curly brackets or the equals sign `=`.

There is another variant that has the same effect:

```
std::string name = {""};
```

Use this notation if you want to perform the initialization with an *initialization list*; in this case, this is a list with only one element—namely, the empty string `""`. As the `=` is optional during initialization, this notation almost always corresponds to `std::string name{""}`. To keep things clear, however, I recommend only using this notation specifically for the initialization list.

6.1.2 Functions and Methods

You have seen that you can create new `string` variables in this way—for example:

- `std::string name;` or `std::string name{};` creates an empty string.
- `std::string name = "value";` or `std::string name{"value"};` creates a string from a literal.

Instead of `"value"`, you can also use a different string—that is, *copy* a different character sequence into a `string`. In the documentation for `string`, you will find further options for initialization and even more functions and methods for `string`. [Table 6.1](#) and [Table 6.2](#) show you the most important ones at a glance.

Function	Description
+	Join two strings together to form a fresh one.
<<	Output of a string.
>>	Read a string until the next whitespace.
getline	Read in a string until the next newline.

Table 6.1 A selection of “string” functions.

Method	Description
length	Length of the content.
at	(Safe) fetch of a single character.
[]	Fetch a single character without checking.
find	Search within the string.
find_first_of	Search for the first character from a set.
substrate	Create a new string from a range.
compare	Comparisons with other strings.
clear	Reset to the empty string.
append	Append a character string.
+=	Alternative spelling for append.
insert	Insert into the string.
erase	Delete part of the string.
replace	Replace a part with another character string.
starts_with	Check if the beginning matches (since C++20).

Table 6.2 A selection of “string” methods.

6.1.3 Other String Types

The `std::string` just introduced is a container for individual characters of type `char`. A `char` (usually) holds eight bits and can therefore only distinguish between 256 states. However, if you want to store Arabic or Chinese characters, for example, or perhaps even Tolkien's dwarf runes, then you need to choose among these options:

- You redefine the meaning of the `char` values from 0 to 255 and interpret them as characters in the foreign language when reading. This is called *encoding*. You must therefore always use the additional information of the *code page* (or *encoding*)—or know it.
- You regard certain characters as special characters (*escape characters*), which mark that a sequence of foreign characters that follows. Simple words (such as English or German) are then written one to one as `char`, but most foreign characters consist of two or more `char` units. This is the case, for example, in the UTF-8 encoding. There, a single character (a *code point*) can be up to six `chars` long.
- You use a different element type than `char`, which can assume more distinguishable states. This is the case with `std::wstring`.

The type `std::wstring` consists of elements of type `wchar_t`. This is 16 bits or 32 bits wide and should be sufficient for foreign characters in the vast majority of cases. (For *all* Unicode characters, even 16 bits are not sufficient, so character strings are then encoded with UTF-16, for example.)

However, because dealing with an element type that has different sizes on different systems complicates program code, there is also `std::u16string` and `std::u32string` from elements of types `char16_t` and `char32_t`. C++20 adds `std::u8string` based on `char8_t`.

You will probably only have to deal with these types when you write international programs. In principle, dealing with these string types is no different to dealing with `std::string` because they are all just synonyms of template class `std::basic_string<>`. If you insert the element type `char`, `wchar_t`, `char16_t`, `char32_t`, or `char8_t` between the angle brackets, you will get the corresponding string types.

This means that all methods that `std::string` has, the others also have. The free functions (e.g., `std::getline()`) have also been written so that they can work on all string types.

However, in practice, it's often not so straightforward to deal with something other than `std::string`. When dealing with external interfaces, you always need to ensure that the correct interpretation is applied: Does the output console expect a specific encoding? What types of strings are expected in dialog boxes? Data streams from the internet are usually `char`-based, so how do you handle them differently?

This is a very broad field. Read up on the system-specific area you need or ask a colleague to explain the extract you need.

6.1.4 For Viewing Only: “string_view”

Strings can be long. Very long. And modern C++ advocates that you program with values instead of pointers or references. This is a crucial issue with potentially huge

objects; it is not wise to use values rigorously with long strings. So what are you to do? The new `string_view` in C++17 is the solution.

A `string_view` (and the corresponding `wstring_view` and the like) is a read-only object into a string. The class offers the same functions and methods so long as they are read-only. So you can iterate with `begin()` and `end()`, find with `find()`, compare with `compare()` or `==`, check the start with `starts_with()`, and much more—and use them in algorithms.

However, you can make certain modifications. For example, you can cut something away from the beginning and end. With `remove_prefix()` and `remove_suffix()`, you can shorten the `string_view` by a number of characters at the front or back.

A `string_view` requires extremely few resources—namely, exactly one pointer into an existing string and its length. Above all, absolutely no heap memory is required. A `string_view` is therefore particularly suitable as a function parameter. Here it is useful that a `string_view` can be easily generated from a string by means of an automatic type conversion.

```
// https://godbolt.org/z/1zfo7fPcs
#include <iostream>
#include <string>
#include <string_view>
void show_middle(std::string_view msg) {           // string_view is a good parameter
    auto middle = msg.substr(2, msg.size()-4);    // substr returns string_view
    std::cout << middle << "\n";
}
int main() {
    using namespace std::literals;
    const std::string aaa = "##Some text##"s;
    show_middle(aaa);                             // conversion to string_view
    auto bbb = "++More text++"sv;                 // string_view as literal
    show_middle(bbb);
}
```

Listing 6.2 A `string_view` can be easily created from a string.

With `show_middle(aaa)`, the compiler automatically converts the string named `aaa` into a `string_view`. It is almost impossible to pass a string as a parameter with fewer resources. As you can see with `msg.substr`, `string_view` also has the method known from string. The return value `middle` is a new `string_view`.

It is worth noting that you can use the suffix `"sv` to generate a `string_view` directly as a literal. `bbb` thus points to the source code (or the static part of the compiled program) and takes up virtually no memory of its own.

You can also sometimes use `string_view` as a return value. However, just as with pointers and references, you must ensure that the original string on which the `string_view` is based still exists.

6.2 Streams

You have seen streams in examples many times, but rarely did their type appear because we used the predefined *streams* of the standard library:

- `std::cout` of type `std::ostream` is used for *standard output*, usually to the console—that is, the screen. Or you can redirect the output to a file.
- `std::cin` of type `std::istream` is used for *standard input*, usually from the keyboard or a redirected file.
- `std::cerr` and `std::clog` are also output data streams of type `std::ostream` and usually end up on the screen, but you can redirect them to a file separately from `std::cout`.

For inputs and outputs of a normal program flow, you normally use `std::cin` and `std::cout`. Each output takes time, and a call has overhead, so `std::cout` is buffered: outputs sometimes do not appear immediately, but are collected first.

Not so with `std::cerr`, whose output should appear immediately. Use this stream for error output. If `std::cout` has been redirected to a file, this output will still appear on the console, as shown in [Listing 6.3](#).

```
// https://godbolt.org/z/P1fn5q8z8
// Call this program with 'prog.exe > file.txt', for example.
#include <iostream> // cout, cerr
int main() {
    std::cout << "Output to cout\n"; // is output after 'file.txt'
    std::cerr << "Error message!\n"; // still appears on the console
    std::cout << "Normal output again\n"; // back into the file
}
```

Listing 6.3 The error stream is separated from the standard output.

6.2.1 Input and Output Operators

As you have often seen, you can use the `<<` operator to output data to `cout`. All built-in data types can be output in this way. You do not have to worry about the format—but you can do so, as you will see in a moment with the *manipulators*. This is worth mentioning because, unlike in the C function `printf()`, the compiler selects the correct format for you—namely, the correct overloading of the global operator `<<`. You can also add overloads for your own data types (see [Chapter 12, Section 12.11](#)).

You can chain several << calls in a row; you just have to name output the stream first. Each << call returns the stream as a result so that it can receive an output again.

The same applies to input via >>: the built-in data types are always read up to the next whitespace (e.g., a space or line feed).

Let's expand [Listing 6.1](#) with a few additional inputs and outputs.

```
// https://godbolt.org/z/xYvPxbjr5
#include <iostream> // cin, cout for input and output
#include <string>
#include <array>
using std::cin; using std::cout; // abbreviations cin and cout
void input(
    std::string &name,
    unsigned &birthDay,
    unsigned &birthMonth,
    unsigned &birthYear,
    long long &socialSecurity,
    std::array<int,12> &monthIncomes) // array is a container
{
    /* input still without good error handling... */
    cout << "Name: ";
    std::getline(cin, name); // getline takes input stream and string
    if(name.length() == 0) {
        cout << "You have entered an empty name.\n";
        exit(1);
    }
    cout << "birth day: "; cin >> birthDay;
    cout << "birth month: "; cin >> birthMonth;
    cout << "birth year: "; cin >> birthYear;
    cout << "social security: "; cin >> socialSecurity;
    for(int m=0; m<12; ++m) {
        cout << "income month " << m+1 << ": "; // multiple outputs
        cin >> monthIncomes[m]; // read in with operator
    }
    cout << std::endl;
}

int main() {
    std::string name{};
    unsigned day = 0;
    unsigned month = 0;
    unsigned year = 0;
    long long ssid = 0;
    std::array<int,12> incomes{};
```

```
    input(name, day, month, year, ssid, incomes);  
    // ... calculations ...  
}
```

Listing 6.4 A function with inputs and outputs.

Here you can see a lot of outputs via `std::cout <<...`. No matter whether it is a string like "birth day: " or a number like `m+1`, you always simply use `<<`. With `cout << "income month "...`, you can also see how several outputs are chained together.

Input from the keyboard is just as easy: the `>>` operator, followed by the variable you want to read into, allows the user to enter a value. However, a little caution is required here: entries should always be completed by pressing the key. For the built-in data types, the next whitespace—such as or —ends the input for the variable. If a space occurs in the user input, everything that follows it is held back for the next output operation via `>>`, and everything gets mixed up.

6.2.2 “getline”

This is also the reason why I do not use `>>` for `name` in `std::getline(cin, name);` but instead use `getline()`. This function reads not only up to the next whitespace, but up to the next end of line—and therefore also allows names with spaces.

6.2.3 Files for Input and Output

When you call a program, you can “redirect” `cout` to a file with `> file`. The following program converts the arguments to uppercase and outputs them:

```
// https://godbolt.org/z/a6qqbWKMs  
#include <cstdlib>  
#include <iostream>  
  
int main(int argc, char* argv[]) {  
    for(int i=1; i<argc; ++i) { // start at 1  
        for(char* p=argv[i]; *p!='\0'; ++p) {  
            char c = toupper(*p);  
            std::cout << c;  
        }  
        std::cout << ' ';  
    }  
    std::cout << '\n';  
}
```

The loop with `i` runs over all arguments that you give the program, and the loop with `p` runs over all characters of the individual arguments. Each argument ends at `\0`. The

`toupper()` function converts a `char` into an uppercase letter. This is how you call the program (do not enter the `$` prompt):

```
$ .\caps.exe Hello text
```

This is what it writes on the screen:

```
HELLO TEXT
```

You can redirect this output:

```
$ .\caps.exe Hello text > output.txt
```

Instead of seeing the output on the screen, it is written to `output.txt`. This procedure is more than common under Unix. For Windows users, it takes some getting used to. But as you are now a C++ programmer and know how to use a text editor, you should have no difficulty using the command line.

You can view the `output.txt` file in any text editor—even in the C++ development environment. Under Unix, write `./program` instead of `.\program.exe`, of course.

If you want to read from a file, you can use `< file`:

```
$ .\program.exe < input.txt
```

However, if you do not want to leave it to the caller to select the file for reading or writing, or if you need more than one input and output, you can create new data streams in the program.

Create a new file with the type `std::ofstream` and provide the desired file name. Instead of `<iostream>`, include `<fstream>`, where the file streams are defined:

```
// https://godbolt.org/z/McqfcWb8j
#include <fstream>
int main(int argc, char* argv[]) {
    std::ofstream myOutput{"output1.txt"};
    myOutput << "line 1\n";
    myOutput << "line 2\n";
}
```

The file is closed as soon as the variable `myOutput` is no longer valid—for example, when its block is exited.

The same applies to reading existing files, except that you use `ifstream` for this:

```
// https://godbolt.org/z/P9z8n8v16
#include <fstream>
int main(int argc, char* argv[]) {
    int value = 0;
```



```
std::ifstream myInput{"input1.txt"};
myInput >> value;
}
```

If the file does not exist, this will not work. You can check whether opening the file has caused an error by applying the nonoperator `!` to the stream.

```
// https://godbolt.org/z/ssW3eKEGq
#include <iostream> // cerr
#include <fstream>
int main(int argc, char* argv[]) {
    int value;
    std::ifstream myInput{"input1.txt"};
    if(!myInput) {
        std::cerr << "Error opening file!\n";
    } else {
        myInput >> value;
    }
}
```

Listing 6.5 Use the `!` operator to check the state of the stream.

6.2.4 Manipulators

If you do not like the format of the output of the standard data types, you can influence it in many ways. In the `<iomanip>` header, you will find all kinds of *manipulators* that you can simply apply to the stream to change the behavior.

This is particularly interesting for floating-point numbers in order to influence the number of decimal places in the output. [Listing 6.6](#) gives a short example.

```
//https://godbolt.org/z/EcnWz3r3j
#include <iostream>
#include <iomanip> // fixed, setprecision
#include <format> // C++20
using std::cout; using std::format; // abbreviation cout, format
int main() {
    cout << std::fixed // dot notation, not scientific
         << std::setprecision(15); // 15 decimal places
    cout << 0.5 << "\n"; // output: 0.500000000000000*
    cout << std::setprecision(5); // 5 decimal places
    cout << 0.25 << "\n"; // output: 0.25000
    cout << format("{:0.4f}", 0.75) << "\n"; // (C++20) output: 0.7500
    return 0;
}
```

Listing 6.6 Use stream manipulators from `<iomanip>` to influence the format of the output.

For the complete list of manipulators, see [Table 6.3](#). You will find a more detailed description of the manipulators in [Chapter 27, Section 27.5](#). Since C++20, you also have the option of formatting with `std::format`. The formatting specifications are easier to read and are familiar from other languages such as Python (or C, for that matter).

Identifier	Description
Header <iostream>	
<code>[no]boolalpha</code>	Textual or numeric bool
<code>[no]showbase</code>	Numbers with or without prefix
<code>[no]showpoint</code>	Floating point always with period
<code>[no]showpos</code>	Display positive numbers with +
<code>[no]skipws</code>	Skip leading whitespaces when reading
<code>[no]uppercase</code>	Capital letters for some output
<code>[no]unitbuf</code>	Flush after every output?
<code>left right internal</code>	Where should fill characters go?
<code>dec hex oct</code>	Output numbers with a different base
<code>fixed defaultfloat scientific hexfloat</code>	Output format of floating-point numbers
Header <iostream>	
<code>ws</code>	Consume all whitespaces
Header <ostream>	
<code>ends</code>	Outputs <code>\0</code>
<code>flush</code>	Output buffered output immediately
<code>endl</code>	Outputs <code>\n</code> and flushes
Header <iomanip>	
<code>[re]setiosflags</code>	(Reset) specified I/O flags
<code>setbase</code>	Output integers in a different base
<code>setfill</code>	Change fill character
<code>setprecision</code>	Output accuracy of floating points
<code>setw</code>	Set the width of the output
<code>put/get_money/time</code>	I/O of special number formats
<code>quoted</code>	Quotation marks for input and output

Table 6.3 The stream manipulators.

6.2.5 The “endl” Manipulator

Not all manipulators actually change the read or write format. The most important exception is `std::endl`. It acts like an end-of-line character `\n`, but has the additional effect of emptying the write buffer—by executing pending write tasks. This takes time; after all, buffering was invented specifically to speed things up. Therefore, if possible, it is better to use `\n` for a line feed.

You should only use `std::endl` if you want to be sure that a screen output is received by your users at a specific time.

If you only want to output the retained buffer and no additional line feed, use the `flush` manipulator—for example, `cout << flush`.

Before a file is closed—that is, before the program ends or the stream variable becomes invalid—you do not need an additional `flush` or `endl`. Closing the file automatically empties the buffer. Only if your program crashes could unwritten remnants remain in the buffer.

6.3 Container and Pointer

Containers are an important part of the standard library. Because they play a central role and also take up a lot of space in it, I will go into them in detail later in [Chapter 24](#). Here I will give you a brief overview so that you know which wheels you don't have to reinvent.

6.3.1 Container

So far, you have encountered types that can hold a single piece of data. For example, if you wanted to store two `int` values, you would have needed two variables (such as `int x, y;`). With `array<int>`, you have already had a taste of one container, and I will explain the background now.

What do you do if you want to store a lot of values? Do you maybe want to iterate over all the values in a loop? Or perhaps you don't know beforehand how many values there will be? For this, the C++ standard library provides *containers*.

Remember that containers are a fundamental idea in the C++ standard library, akin to a concept before this term became a language element in C++20. The respective interfaces of all containers are the same, and you can apply what you know about one container to all others—provided you follow some rules. There are exceptions, and quite a few of them, but once you understand the idea, you will master all containers.

There are many containers in the standard library, and depending on the purpose of the application, one or the other is the sensible choice. In this chapter, I will focus on two particular containers because they are fundamentally different from each other.

6.3.2 Parameterized Types

All containers have in common that they are *parameterized types*. They consist of a main type—the actual container—and the type or types that are put into the container.

The main type is followed by parameters in angle brackets, for example, `vector<int>`, `map<string,Car>`, or `array<double,10>`. It is extremely important that you consider the entire construct as a *single type*. A `vector<int>` is different from a `vector<long>`, just as an `int` is different from a `long`. This is crucial for function overloading and for operator resolution. For example, you can write two functions like this:

```
long sum(vector<int> arg);
long sum(vector<long> arg);
```

This works because the argument types are different. You are writing a normal overload (see [Chapter 7, Section 7.8](#)).

This also applies to (constant) values as parameters; for example, `array<int,10>` is a different type than `array<int,11>`.

As a consequence of the fact that *maintype<parameter>* as a whole forms the type, this must already be determined at compile time. This is particularly noticeable with `array`. You might be tempted to do one of the following things:

```
void calculateImArray(int n) {
    array<int,n> data {};          // ✗ n must be constant
    // ...
}
or
long sumArray(array<int,n> data) { // ✗ where does this n come from?
    int sum = 0;
    for(int e : data)
        sum += e;
    return sum;
}
```

Neither is possible, as `n` is a variable that is only determined at runtime. At most, you can use simple calculations such as `3+4` or a `constexpr`.

But enough of the preview of `array` as a vehicle for the language feature of parameterized types. I'd better get specific now.

Omission of the Element Type

Note that with C++17, you can often omit the type parameters between the angle brackets. The compiler can determine the element type of the container using the constructor arguments:

```
std::vector vec { 1, 2, 3 };
```

This is still a `vector<int>`, except that you can save yourself some typing. However, this is not always sensible (or possible):

```
std::vector<std::string> elves { "Elrond", "Galadriel" };
using namespace std::literals;
std::vector dwarves { "Oin"s, "Gloin"s };
```

With the suffix `"s`, you force string constructor arguments and get a `vector<string>` also for dwarfs.

6.4 The Simple Sequence Containers

■ `std::array`

You specify the number of elements this container should hold at creation. It neither grows nor shrinks; you access its elements with an index or iterate over ranges.

■ `std::vector`

This all-rounder stores its elements consecutively. You access its elements via a numeric index or iterate over ranges. In addition, it grows automatically when you add elements.

I strive not to use the ambiguous term *arrays*, but always write either *array* for `std::array` or *C-array* for `typ[]`. In other sources, the latter is usually simply called *array*—or, in some texts, *field*.¹

6.4.1 “array”

You use the `array` when you know in advance how many elements you need. As you can see in the first line of `main()`, you specify the number of elements in the declaration. It cannot be changed later.

```
// https://godbolt.org/z/cd1d1W7T1
#include <array>
#include <iostream>

using std::cout; using std::array; using std::string;

int main() {
    array<string,7> wday = { "Monday", "Tuesday",           // define
                           "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
}
```

¹ Other texts use *field* for `vector`. Because of all this confusion, I prefer to use the original terms of the C++ language in this book.

```

cout << "The week starts with " << wday[0] << ".\n"; // read values
cout << "It ends with " << wday.at(6) << ".\n";      // read safely
/* nothern? */
wday[5] = "Satdee";                                // change values
}

```

Listing 6.7 You store a fixed number of elements in an “array.”

You access the elements with square brackets with `wotag[0]` and `wotag[5]` via a number index; you can both read and write values. The alternative method `at(index)` is somewhat safer: while with `[]` there is no check for whether you have used an index that is too large or too small, and your program may crash or worse (it continues to run with incorrect values), `at()` includes a check with which you can catch an error—or at least receive an error message before the program ends (see [Chapter 10](#)).

The number of elements in the declaration of `wday` must be a constant. You cannot use a variable here whose value you have previously determined. As a number literal, 7 is of course constant. However, it is good style to declare such a number as a constant beforehand.

```

// https://godbolt.org/z/zfe7dM4ex
#include <array>
#include <iostream>
constexpr size_t MONTHS = 12; /* months in year */

int main() {
    std::array<unsigned,MONTHS> mdays = {           // okay with a constant
        31,28,31,30,31,30,31,31,30,31,30,30,31};
    unsigned age = 0;
    std::cout << "How old are you? "; std::cin >> age;
    std::array<int,age> years of life;              // ✗ array size not by variable
}

```

Listing 6.8 The array size must be constant.

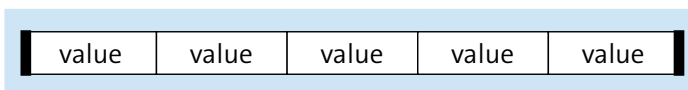


Figure 6.1 An “array” can neither grow nor shrink. Its elements are arranged compactly.

If you want to pass an array as a parameter to a function, its type must exactly match the array definition. Because `array<int,4>` is a different type than `array<int,5>` and `array<short,4>`, it would not match such a parameter type. To avoid repeating yourself constantly, you should give the array type its own name using a `using` (or `typedef`) statement beforehand.

```
// https://godbolt.org/z/a4zxbd99z
#include <array>
#include <algorithm>           // accumulate
#include <numeric>             // iota
using January = std::array<int,31>; // alias for repeated use

void initJanuary(January& jan) { // the exact array as parameter
    std::iota(begin(jan), end(jan), 1); // fills with 1, 2, 3 ... 31
}
int sumJanuary(const January& jan) { // the exact array as parameter
    return std::accumulate(begin(jan), end(jan), 0); // helper function for sum
}
int main() {
    January jan; // declares an array<int,31>
    initJanuary( jan );
    int sum = sumJanuary( jan );
}
```

Listing 6.9 If you need to use an array definition more than once, use “using.”

If you need to store a fixed number of uniform elements, an `array` is ideal. It serves well especially as a member variable, and also for static data that you type literally into the source code.

Its particular strength is that the data lies directly next to each other in memory, which is useful in many application areas; for example, it can be written to the hard drive particularly quickly as a block.

Otherwise, you can see the useful `accumulate` and `iota` functions from the `<algorithm>` and `<numeric>` headers, both of which are so-called algorithms. These are auxiliary functions that you can apply to many, if not all, containers. You will learn more about them in other parts of the book. In [Chapter 25, Section 25.6](#), they are all listed in detail.

6.4.2 “vector”

However, it is impractical to always have a predetermined number of elements in the container. The all-around talent of the container world is undoubtedly the `vector`. Like an `array`, it only contains elements of one type. Access also works in the same way—namely, via a number index (starting at 0) or via an iterator.

However, the size of the `vector` is dynamic: it can grow as needed. For example, you can initialize it empty and then add as many elements as your computer’s memory allows:

```
// https://godbolt.org/z/T9jsEoMj4
#include <vector> // You need this header
int main() {
    std::vector<int> squares{}; // initialize empty
    for(int idx = 0; idx<100; ++idx) {
        squares.push_back(idx*idx); // append an element
    }
}
```

With `push_back()`, you append an element to the end of the vector. Alongside index access, this is one of the most useful functions of `vector`—for example, `squares[idx]`.

In a vector, the elements are stored consecutively in computer memory. This is CPU-friendly and makes it extremely fast when accessing from front to back. However, the vector has the following disadvantage: if you want to insert an element in the middle or at the front, all elements to the right of it are shifted one position to the side—and this is usually a time-consuming process.

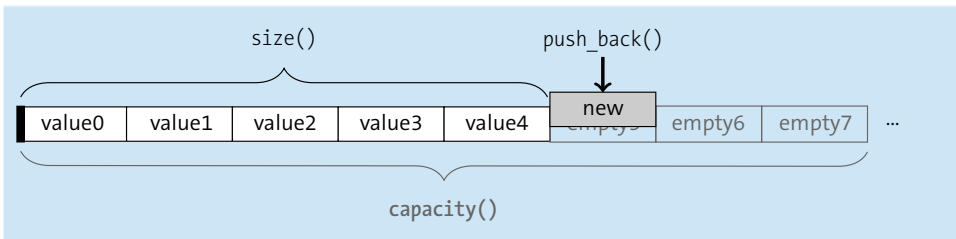


Figure 6.2 Schematic representation of a “vector.”

Prefer to Insert Elements at the Back of a “vector”

Whenever you can, you should only insert elements at the back of a vector. The methods for this are `push_back()` and `emplace_back()`.

There are several options available for operations on all elements of a vector. You are already familiar with the range-based `for` loop.

```
// https://godbolt.org/z/hhhavG5he
#include <vector>
#include <iostream> // cout, endl
int main() {
    std::vector squares{1,4,9,16,25}; // initialize filled
    for(int number : squares) // number is one square at a time
        // ...
}
```



```
        std::cout << number << " ";
    std::cout << std::endl;
}
```

Listing 6.10 The simplest iteration uses a range based “for” loop.

Here, all elements are output sequentially: 1 4 9 16 25. Alternatively, you can also access the elements by index. Don't forget that the counting (almost always in C++) starts at zero. You can use `size()` to check how many elements are in the vector.

Note that I wrote `std::vector` here instead of `std::vector<int>` because the compiler deduces `<int>` from the constructor arguments since C++17.

```
// https://godbolt.org/z/xcKM7v1Eo
#include <vector>
#include <iostream>                                // cout, endl
int main() {
    std::vector qus{1,4,9,16,25};
    for(unsigned idx=0; idx<qus.size(); ++idx) // size returns the count
        std::cout << qus[idx] << " ";        // [idx] or at(idx) gets an element
    std::cout << std::endl;
}
```

Listing 6.11 Accessing the elements by index.

In C++, however, you should use iterators instead of an index. They can be used more widely, may be easier for the compiler to handle, and, above all, are the same for all containers.

```
// https://godbolt.org/z/1WG8zvYhYe
#include <vector>
#include <iostream>                                // cout, endl
int main() {
    std::vector qus{1,4,9,16,25};
    for(auto it = qus.begin(); it!=qus.end(); ++it) // between begin() and end()
        std::cout << *it << " "; // with *it you get from the iterator to the element
    std::cout << std::endl;
}
```

Listing 6.12 The use of iterators for a loop.

Just like containers, *iterators* are a general idea. They work closely with containers and are also explained in detail in [Chapter 24](#). For now, I will summarize them for you in the shortest possible way:

- `qus.begin()` and `qus.end()`
Return iterators for the start and end of the container, similar to how `0` and `size()` provide the bounds of the indexes.
- `auto it = qus.begin()`
Defines an iterator and initializes it to the start of the container.
- `it != qus.end()`
Checks if you are at the end. Always check with “not equal,” never with “less.”
- `*it`
Goes from the iterator to the element. You “dereference” the iterator.

The Standard Library Contains Many Containers

The other containers and plenty of instructions on how to use them can be found in [Chapter 24](#).

6.5 Algorithms

In addition to the capabilities of the containers, there are *algorithms*. Most of them can be found in the `<algorithm>` header. There is a long list of auxiliary functions that help you to solve specific problems using general methods.

All these algorithms consistently work on iterators, which is why they work on (almost) all containers. It is important to know that the power of the containers is not exhausted by their methods.

In the following example, a vector is filled with an algorithm, and then elements meeting a specific criterion are counted.

```
// https://godbolt.org/z/4G67rso8T
#include <vector>
#include <algorithm>           // count_if
#include <numeric>             // iota
#include <iostream>
bool even(int n) { return n%2==0; } // test for even
int main() {
    std::vector<int> data(100); // 100 x zero
    std::iota(data.begin(), data.end(), 0); // 0, 1, 2, ... 99
    // counts even numbers
    std::cout << std::count_if(data.begin(), data.end(), even);
}
```

Listing 6.13 Counting with an algorithm.

6.6 Pointers and C-Arrays

In discussing the built-in data types, I have so far omitted two very important types: pointers and C-arrays. These two are closely related, which is why I treat them together. There are two reasons that this happens so late:

- They have their own principles and bring with them a huge range of things that you need to understand to use them effectively.
- For the tasks they solve, they are only the second-best option in modern C++. Since C++11 at the latest, there is usually a better alternative in the language or the standard library.

Therefore, I will give you only a brief introduction to pointers and C-arrays. You will learn more in [Chapter 20](#)—and you will also learn more there about the alternatives.

6.6.1 Pointer Types

A *pointer* is the C variant of a reference, which is still used frequently in C++ due to historical reasons. It is an indirection to the actual value. Therefore, every type can also exist as a pointer—even pointers themselves, meaning that a pointer can point to a pointer. A pointer represents an address in memory, and the computer can perform calculations with it: it can form differences, increment, and so on. Therefore, pointers are well-suited for large and dynamic memory amounts. With the address operator `&`, you determine the address of an object and obtain a pointer type. You can recognize it by an asterisk `*` following the type. For example, after `int x=12;` you can determine the address with `int* p = &x;`. This `int*` points to an `int` value, and a `char*` points to a `char` value. Only the pointer type `void*` is flexible in what it points to—and is therefore unsafe and should be avoided. If a pointer points to nowhere, it has the special value `nullptr`.

6.6.2 C-Arrays

C-arrays are denoted with square brackets `[]`. In the declaration, these are placed after the variable name. For example, `int nums[10]` stores 10 `int` values directly next to each other. The type of `nums` here is `int[10]`—so long as `nums` is not passed to a function. Unfortunately, the compiler cannot pass the C-array size itself, which is why the C-array then “decays” into a pointer. Thus, the size-less notation `int[]` is equivalent to `int*`. Among other reasons, due to this major disadvantage, you should always resort to alternatives if possible: `string`, `vector`, or the like for dynamic data pieces, or `array`, `pair`, or `tuple` for fixed pieces.

Chapter 7

Functions

Chapter Telegram

- **Declaration and definition**

The *declaration* only introduces an identifier and provides the compiler information, such as the type. In contrast, the *definition* also reserves space for the construct. With functions, this is the difference between naming the function head *without* a function body or *with* one.

- **Side effect**

A side effect occurs when a function changes the state of the program by means other than parameters or return values.

- **Function call**

This is the interruption and later continuation of the current program flow by the body of a function.

- **Function header**

This is a function declaration, consisting of the return type, function name, and parameters.

- **Function body**

These are the statements belonging to a function definition.

- **Return value and return type**

These are what the function returns with a `return` statement.

- **Parameters**

A parameter consists of a parameter type and an optional parameter name and is part of the function declaration. The parameter name can be used to access the passed parameter inside the function body.

- **Call-by-value**

This is a parameter that is copied into the function as a value for the function call. It is part of the parameter type in the function declaration. Changes to the parameter within the function have no effect outside.

- **Call-by-reference**

This is a parameter that is declared in such a way that the function accesses the passed construct directly by reference—for example, a variable—and can thus potentially change its value outside the function.

- **Free function**

A free function is a function that is not part of a class. It is defined globally or in a namespace.

- **Method**

A function that is called with a specific instance of a class is a method. In C++ these are declared inside their class (details in a later chapter).

- **Default parameters**

These are predefined values for calling a function for optional arguments.

- **Overloading functions**

These are multiple functions with the same name that differ in the types of parameters and/or return.

You have already seen some functions, and you have already written some of them yourself—primarily the `main()` function, because you can't do without `main()`. In this chapter, I want to explain the possibilities you have with functions, especially when you write them yourself.

A function is something that you can call from another location. Functions can also have a return value. A function that does not have a return value is said to return `void`. In addition to calculating the return value, parameters can also be changed if you allow it. A function can also have *side effects*, meaning it can additionally change other objects in the program.

7.1 Declaration and Definition of a Function

When you write a function and immediately specify what its program code looks like—that is, with the instructions in the curly brackets—then this is the *definition* of the function:

```
int addTwo(int a, int b) {  
    return a + b;  
}
```

However, you can also first tell the compiler that a certain function exists somewhere; where exactly, it should figure out later. Then you omit the brackets with the statements and only name the function header—and you have a *declaration*:

```
int addTwo(int a, int b);
```

When you write a program that uses `addTwo`, it is initially sufficient that you have already made a declaration of this function. Only when linking the program must the corresponding definition be present somewhere in one of the program parts. For the

declaration, it is even sufficient if you only name the parameter types; you can omit their names:

```
int addTwo(int, int);
```

In general, I would not recommend this as the names already provide important information to readers.

Declarations are often found in header files of modules (*.h or similar). The corresponding definition and thus the implementation can then be found in a *.cpp file (or a file with a different extension) or in a precompiled library.

7.2 Function Type

Just like almost everything else in C++, a function has a *type*. This type determines how you call the function and what you can do with the result. The type of a function consists of the types of the parameters and the return type.

So the functions

```
bool print(std::string arg1, int arg2);
```

and

```
bool formatValueIntoString(std::string format, int value);
```

have the same type. The exact type is something composite—in this case:

```
std::function<bool(std::string, int)>
```

You see that in this notation, the return type `bool` is directly behind the angle bracket, and the parameter types are in parenthesis inside the angle brackets. `function` is just a tool from the standard library and my preferred C++ notation, which I understand better than the *actual* type in C form—that is:

```
bool (*)(std::string, int)
```

This is where a pointer `*` comes into play, because a function type actually represents a pointer. You can find out more about these two different notations in [Chapter 23](#). For now, you should understand that there is a representation of the type of functions and that this type depends on the parameter types and the return type.

You will see later that this is important because you can store functions in C++ just like `int` and `string` values in variables, pass them as parameters, or return them from functions.

7.3 Using Functions

Before we delve further into how to write functions yourself, let's first look at their usage.

```
// https://godbolt.org/z/TdrebYodx
#include <iostream>           // cout
#include <cmath>              // sin
#include <string>
#include <vector>
using std::sin;
int main() {
    std::cout << "sin(0.0): " << sin(0.0) << "\n";           // call sin() with literal
    double angle = 3.1415/2;
    std::cout << "sin("& << angle << "): " << sin(angle) << "\n"; // call with variable
    std::string name = "Han Solo";
    std::cout << name.length() << "\n"; // call a method
                                           // ... conceptually like length(name)

    std::vector<int> data{};
    data.push_back(5);                    // further method call with parameter
    data.push_back(10);
    std::cout << data.back() << " ";
    data.pop_back();
    std::cout << data.back() << "\n";
    data.pop_back();
}
```

Listing 7.1 Use functions.

The function `sin()` is called once with the literal `0.0`—as a literal also a constant value—and once with the variable `angle`. As you would expect from the mathematical function *sin*, the result of a calculation is returned here.

You can see that the variable with which I call `sin()` is of type `double`. That fits, because in the standard library the function is declared as `double sin(double)`: it takes a `double` and also returns one. Parameter types must match each other when called: you could not have called `sin("Han Solo")`, because `sin()` is not intended for character strings. There are exceptions where a conversion takes place; you will get to know them and write some of them yourself.

Note that I am also presenting *method calls* here, such as `name.length()` and `data.push_back(5)`. The main difference is that a function like `sin()` is “free” (a *free function*), but `length()` is firmly associated with the class `std::string`. When you call a function using *variable.functionname*, this is the special form of the *method*. This term comes from object orientation and is discussed in detail in the chapter on classes (see [Chapter 12](#)).

Note here that `length()`, `push_back()`, `back()`, and `pop_back()` are also constructs that you use like functions (and define later) and that you constantly will make use of.

7.4 Defining a Function

In general, you define a function as follows:

```
ReturnType FunctionName ( ParameterType ParameterName , ... ) { FunctionBody }
```

Or you can use the newer syntax, where the return type is at the end or can be omitted altogether. I will go into this more detail in [Section 7.11](#). Sometimes it can be clearer:

```
auto FunctionName ( ParameterType -name , ... ) [ -> ReturnType ]  
{ FunctionBody }
```

You have already heard some of this in passing, but I would like to say something about all these elements once again:

■ Return type

The return type is the type of a value that you return from the function with `return`. All expressions in the `return` statements must be of this type or convertible to this type. If the return type is `void`, you do not need to specify a `return`. If you do, then write `return;` without a value. At the point where you then use the function, you can use the return value in an expression of the appropriate type—for example, in a calculation or an assignment. You can see some examples of return types in [Listing 7.2](#).

■ Function name

This is a regular identifier under which you will later use the function. It may be that your function belongs to a class (method), in which case the function name is made up of `ClassName::MethodName`. Also, when you get to know function templates later, the name may be followed by angle brackets `<...>`. I will cover this in [Chapter 23](#).

■ Parameters

If your function does not take any parameters, the parentheses are empty `()`. Otherwise, the types and names of the function parameters are listed here, separated by commas.

■ Parameter type

Each parameter must be specified with at least its type. The abundance of possible types ranges from simple ones (such as `int` and `std::string`) to complex ones with several angle brackets (such as `map<string, vector<int>>`). It is particularly important to distinguish whether you want to specify a parameter as a *reference* or *pointer*; in the latter two cases, an `&` or `*` is part of the type. If the parameter cannot be modified within the function, `const` is part of the type.

■ Parameter name

To be able to use the parameter in the function at all, refer to it by the name given

here. You can omit it if you are only *declaring* the function here (and not *defining* it) or if you do not use the parameter in the function at all. However, I recommend using a clear, descriptive name so that the function header becomes meaningful.

■ Function body

The function body (together with the surrounding curly brackets) consists of statements: none, one, or many of them, as you see fit. If the function has a return value, there must be at least one `return` statement here.

```
int func();                // returns an int
std::string func();        // a string from the standard library
void func();               // no return value
std::pair<int, std::string> func(); // composite type from the stdlib
vector<int> func();         // returns a new container
vector<int>& func();         // reference to some container
const vector<int>& func();    // same, but you cannot change it
```

Listing 7.2 Different return types of functions.

C++20: Abbreviated Function Template

As of C++20, you can use `auto` as the parameter type. You then define a function template without the cumbersome notation with `template` and angle brackets. The compiler turns

```
void func(auto a) { ... }
```

into

```
template<type name A> void func(A a) { ... }.
```

You can read more about templates in [Chapter 23](#). I mention it here because on the one hand, this notation does not reveal the template; on the other hand, it offers great opportunities for versatile yet compact code.

7.5 More about Parameters

In C++, the declaration of the parameter in the function header determines how the parameter behaves when called. The most important distinction is whether the parameter is passed as a *value* or as a *reference*.

7.5.1 Call-by-Value

If you do not take any special precautions, the parameters of a function in C++ are passed *as values* (*call-by-value*). This means that if a variable `x` currently has the value 5 and you pass `x` to a function, then the 5 ends up in the function—not the `x`.

```
// https://godbolt.org/z/8zcvMdM3b
#include <iostream>

void print_val8(int n) {           // parameter as value
    std::cout << n << " ";
    n = 8;                        // sets parameter to 8
    std::cout << n << "\n";
}

int main() {
    int x = 5;
    print_val8(x);                // x as value: prints 5, then 8
    std::cout << x << "\n";      // x remains unchanged at 5
    print_val8(42);               // 42 as value: prints 42, then 8
}
```

Listing 7.3 Parameters are initially passed as a value.

When `print_val8` is called, the compiler copies the *value* of the arguments to the position `n` for the function. Therefore, the change at `n = 8;` has no effect on the variable `x` with which `print_val8(x)` was initially called.

Call-by-Value for Structures with Pointers

At this point I must warn you that when interacting (especially) with pointers, things get a little more complex. If a pointer `*` is part of a structure or class that you pass by value, only the pointer is copied in this structure, not what it points to. This means that if you change what the pointer points to, you also change the original.

7.5.2 Call-by-Reference

If you want the assignment to affect the outer variable, you need to declare the function in C++ in such a way that it receives the parameter *as a reference* (*call-by-reference*). To do this, use the ampersand `&`.

I will vary the previous example for comparison before showing you the meaningful use.

```
// https://godbolt.org/z/63dTshz3Y
#include <iostream>
void print_ref8(int& n) {         // parameter as reference
    std::cout << n << " ";
    n = 8;                       // sets parameter to 8
    std::cout << n << "\n";
}
```

```
int main() {  
    int x = 5;  
    print_ref8(x);           // x as reference: prints 5, then 8  
    std::cout << x << "\n"; // x is now 8  
}
```

Listing 7.4 Add an ampersand to make it a reference parameter.

The function `print_ref8` corresponds exactly to `print_val8` from the previous listing, except that in this example the parameter of `print_ref8` is declared with the `&` as a reference—namely, `int& n`. Therefore, when calling `print_ref8(x)` in `main()`, the compiler does not copy the value of `x` into the function, but passes the `x` itself *as a reference*; within the function, it is then addressed with the parameter name `n`. This way, you can call `print_ref8` from completely different program locations and pass completely different variables; within `print_ref8`, they are always called `n`.

There is another difference: you cannot use `print_ref8(42)`. Can you see why? Within `print_ref8`, the `42` would be referred to by the name `n`—and later an assignment is made in `print_ref8` with `n = 8`. This means it would be `42 = 8`, and that obviously is not possible. However, the compiler prevents this not only at the assignment `n = 8`, but at the call `print_ref8(42)`: the parameter type is `int&` (read: “reference to `int`”) and not `int`, and the compiler cannot refer to `42` in this way; `42` is not compatible with `int&`, and the compiler will complain.

7.5.3 Constant References

To pass both `x` and `42` to a function that takes a reference as a parameter, you must “promise” the compiler that you will not assign anything to the reference. With `const int& n`, you tell it that `n` is constant inside the function.

```
// https://godbolt.org/z/dcKnx3Pno  
#include <iostream>  
void print_cref(const int& n) { // parameter as constant reference  
    std::cout << n << " ";  
}  
  
int main() {  
    int x = 5;  
    print_cref(x);           // call with a variable  
    print_cref(42);          // call with a constant literal  
}
```

Listing 7.5 You can use constant references as parameters for any call.

Now, of course, the line with `n = 8` is gone, because otherwise you would be breaking your promise. The compiler would now find fault with this line, because you cannot assign anything to a `const int&`, which `n` now is. However, you can use a variable when calling `print_cref8(x)` or a literal `print_cref8(42)`. This means that the following two function declarations can be used equally broadly as far as the arguments are concerned:

```
void print_val8(int n);           // Parameter as value
void print_cref(const int& n);    // Parameter as constant reference
```

7.5.4 Call as Value, Reference, or Constant Reference?

All these variants of declaring a parameter have their justification, and you have to decide which one to use on a case-by-case basis:

- When called *as a value*, a copy is created. This can be a time- and memory-consuming process at program runtime. On the other hand, you can be sure that you will not accidentally change any external values.
- When called *as a reference*, no copy is created, but the compiler transfers the address of the parameter into the function. This is quick and saves space. But do you really want to change externally defined variables? Sometimes, yes, but not always.
- Use a *const reference* if you decide for a reference but do not want the variable changed.
- If you use a *non-const reference*, you simply cannot pass some parameters, just as the literal 42 cannot be passed to an `int&`, for example. There are also constant values and objects, all of which you exclude with an `&` only parameter.

A disadvantage when using references (and pointers) can arise due to *aliasing*. There are cases in which a reference in a function refers to a variable (via an *alias*) that no longer exists; it has already become invalid outside. Program crashes are the result. As the parameter is copied specifically for the function call when it is called as a value, this cannot happen here.

One way to avoid these program crashes is to forego references. Because a typical application is the output parameter, I recommend that you use a return value instead:

```
// https://godbolt.org/z/oh91YvaqT
#include <iostream>
void twice(double &num) {          // output parameter as modifiable reference
    num *= 2.0;
}

int main() {
    double num = 7.25;
```

```
twice(num);  
std::cout << num << "\n"; // now 14.5  
}
```

It may look silly in this simple example, but if you have larger objects, you'll be tempted to do this all the time. Write instead:

```
// https://godbolt.org/z/q4ov41cnv  
#include <iostream>  
double twice(double num) { // value parameter and return value  
    return num * 2.0;  
}  
int main() {  
    double num = 7.25;  
    num = twice(num); // change expressed by return value  
    std::cout << num << "\n"; // also 14.5  
}
```

Here, the parameter `number` is copied when `twice` is called, but this makes the program safer and also clearer. The good news is that copying has improved since C++11, especially for data types from the standard library: values are now often *moved* instead of copied, which is always beneficial for performance.

When in Doubt, Use Value

If in doubt, opt for a value parameter. This must be copied when called; for large objects, you can consider whether a constant reference is more suitable.

7.6 Functional Body

The really exciting thing about a function is, of course, what it does—because that's what a program is all about. I'm not telling you anything new here. You've already become familiar with `main()` as a function, and other functions are not much different.

Take a look at the functions in [Listing 7.6](#). We will need them later, so let's examine how their function bodies are structured now.

```
std::vector<int> primes = {2}; // global variable  
bool isPrime(int n) { // own function  
    for(int factor : primes) { // access to global variable  
        if(factor*factor > n) // access to parameter  
            return true;  
        if(n%factor==0)  
            return false;  
    }  
}
```

```

    return true;
}
void calculatePrimesUpTo(int limit) { // another custom function
    for(int n=3; n<limit; n=n+2) {
        if(isPrime (n)) {           // use custom function
            primes.push_back(n);
        }
    }
}
}

```

Listing 7.6 Various definitions of functions.

First comes the definition of the `isPrim` function. The function receives a parameter of type `int`, which is used within the function under the name `n`. However, variables other than the `n` parameter are used in the function body.

With `for(int factor : ...)`, I introduce a new `int` variable for the loop—a *local variable*. You already know that it is only visible within the `for` loop. You cannot access `factor` from another function.

I also use `primes` in the `for`—a *global variable*: this was declared outside the function and is “visible” within the function. You can access all global variables that were previously declared within this module. This is the case with `primes`.

The same applies to other functions: you can use other functions if they have been declared beforehand. This happens, for example, in `calculatePrimesUpTo` in the `if`: I call `isPrim`, and the compiler already knows the function because I defined it earlier (higher up in this module)—and a definition is even better than a declaration in this regard.

If for some reason I had written the function `isPrim` later in the program code, the compiler would not yet know it and would complain. However, I could remedy this by at least declaring the required function beforehand.

```

// ... extract ...
bool isPrime(int n);           // Declaration of function defined later
void calculatePrimesUpTo(int limit) {
    for(int n=3; n<limit; n=n+2) {
        if(isPrime(n)) {       // Use of function defined later
            primes.push_back(n);
        }
    }
}
}
bool isPrime(int n) {           // Definition after usage
    // ... as before ...
}

```

Listing 7.7 A forward declaration without a function body.

This is sometimes called a *forward declaration*.

When you combine a program from multiple modules (*.cpp files), you can also use functions from another module. In principle, you use exactly the forward declaration mechanism just explained: if you have defined a function `f()` in module A that you want to use in module B, then simply declare `f` identically at the beginning of module B, and the parts will come together when the entire program is linked.

However, this is only the first step toward the correct procedure. For the *.cpp file of module A, you also maintain a corresponding header file *.h that contains all function declarations that are potentially to be used elsewhere. Then, in module B, use an `#include` of this header file and effectively achieve the same result as with the manual forward declaration.

I will demonstrate this cleaner solution in [Chapter 12](#).

7.7 Converting Parameters

When a function receives a parameter with a certain type, there are several possibilities for this parameter when it is called. Let's assume you have declared the `func` function as follows:

```
void func(double param);
```

Then the following can happen when you call it up:

- **func(5.3)—the type matches**

5.3 is a double literal and therefore exactly matches the parameter type double. The call works.

- **func("Text")—the type does not match**

Here, the call is attempted with a `const char[]`, which does not match double. This is an error that the compiler will inform you about.

- **func(7)—the type can be converted**

Although 7 is an int, it can be converted into a double by the compiler. The call works, but the compiler first converts 7 to 7.0.

For the built-in types, the compiler has some rules about which conversions it can perform. Here is a very brief overview:

- int to long or short to int is not a problem during conversion.
- long to int is also done by the compiler, but a dangerous overflow can occur here.
- short, int, or long can be converted to float or double, although large long numbers can lose a few digits of precision when converted to float.
- float or double can be converted to short, int, or long. However, the fractional parts are lost here, and an overflow can occur again if the numbers are too large.

- For a `const char[]`—such as the literal `"Text"`, for example—there is a conversion to `std::string` (this is not provided by the compiler, however, but by the standard library).

The next listing shows some transformations that are possible and the problems you might encounter. However, note that the types can vary in width depending on the system, and you might encounter different transformations. In addition, overflows should strictly be avoided as some behavior can become arbitrary (technical term: *undefined*). The example outputs were generated on a 64-bit Intel Linux system using GCC 4.9.

```
// https://godbolt.org/z/zb1T7Kr3G
#include <iostream>
#include <format>
void prints(short s, int i, float f, double d) {
    std::cout << std::format("short: {} int: {} float: {:.2f} double: {:.2f}\n",
        s, i, f, d);
}
int main() {
    int mill = 1000*1000;           //1 Million
    prints(mill, mill, mill, mill); // short overflows
    // Output: short: 16960 int: 1000000 float: 1000000.00 double: 1000000.00
    long bill = 1000L*1000L*1000L*1000L; //1 Trillion
    prints(bill, bill, bill, bill);    // even int overflows, float becomes inaccurate
    // Output: short: 4096 int: -727379968
    // float:999999995904.00 double: 1000000000000.00
    float three = 3.75f;
    prints(three, three, three, three); // decimal places are lost
    // Output: short: 3 int: 3 float: 3.75 double: 3.75
}
```

Listing 7.8 Some conversions take place here. The conversions may look different on your system.

In `prints(mill,...)`, the parameter `short s` overflows during the transformation of the value 1000000 (1 million), and the content is meaningless. The transformation to a `float` or `double` occurs without issues. `format` has been available since C++20; before that, you can use `std::cout << s ...`

The even larger value of 1 trillion also causes the `int` to overflow with `prints(bill,...)`. The conversion to a `float` does avoid overflow, but the `float` does not have enough precision to store the exact value.

No integer type, such as `short` and `int`, can hold a decimal number like `3.75f`. When converting in `prints(three,...)`, the decimal places are simply truncated. However, a

float can be converted to a double without loss. The other way around, information could be lost.

Outlook: Conversion of Own Types

If you later define your own types, you can decide for yourself which conversions to or from your type are permitted. You can convert a type A *into* your type MyType by defining a constructor MyType(A). And MyType can be converted to type B if you define a method operator B() in MyType.

7.8 Overloading Functions

If you have defined the function

```
void print(int value) {  
    std::cout << "int value: " << value;  
}
```

but do not want print(3.75) to output a nonsensical int value: 3, you can also define the print function for a parameter of type double. If a function with the same name is defined multiple times, it is called *overloading*.

```
// https://godbolt.org/z/9YcPraT15  
#include <iostream>  
void print(int value) { std::cout << "int value: " << value << " "; }  
void print(double value) { std::cout << "double value: " << value << " "; }  
void print(int v1, double v2) { std::cout << "Values: "<<v1<< ", "<<v2<< " "; }  
int add(int n, int m) { return n + m; }  
double add(double a, double b) { return a + b; }  
int main() {  
    print( add(3, 4) );           // add(int, int) and print(int)  
    print( add(3.25f, 1.5f) );   // add(double, double) and print(double)  
    print( 7, 3.25 );           // print(int, double)  
}
```

Listing 7.9 The “print” and “add” functions have been overloaded for multiple types.

You see that print has three overloads. If you call it with an int, then print(int value) is called, with a double, of course: print(double value). You can also define overloads with a different number of parameters, as you see in the third print variant. The call to print(7, 3.25) fits this.

When choosing the right overload, the compiler follows a few rules. If the parameters exactly match an overload, it's easy. But if a parameter doesn't match exactly, then the compiler has to choose. For example, I wrote:

```
add(3.25f, 1.5f)
```

Now, 3.25f and 1.5f are float literals; if I had written 3.25 and 1.5, they would have been double literals. However, it is clearly better for the compiler to choose the `add(double, double)` variant instead of `add(int, int)`, as nothing is lost when converting from float to double, unlike a conversion from float to int.

If you try `print(1.25, 2.75)`, the compiler obviously cannot choose either `print(int)` or `print(double)`; the number of parameters simply doesn't match. So, it will call `print(int, double)` and convert 1.25 to an int. You will get the output Values: 1, 2.75.

If the compiler doesn't find a suitable overload with the conversions it can perform, it will complain to you with an error.

Sometimes, however, the compiler can find multiple overloads that all fit equally well or equally badly. In this case, the compiler will also respond with an error message, stating that the overload resolution was not unambiguous. If you call

```
add(4, 2.85); // ⚡ Overloading unclear
```

the compiler cannot decide whether to use `add(int, int)` or `add(double, double)`. It will then tell you that this call cannot be resolved unambiguously.

To protect you from surprises, it should be mentioned that while parameter types contribute to overload resolution, return types do not. The compiler will complain about overloads that differ only in return type:

```
// https://godbolt.org/z/58cTfvYhe
int two() { return 2; }      // ⚡ once with int as return type...
double two() { return 2.0; } // ⚡ ... and once with double
int main() {
    int x = two();
    double y = two();
}
```

However, together with distinguishable parameter types, return types can also be different:

```
// https://godbolt.org/z/evGqG1oqa
int twice(int a) { return a * 2; }
double twice(double a) { return a * 2.0; }
int main() {
    int x = twice(7);
    double y = twice(7.0);
}
```

This is widely used in the standard library. For example, the `begin()` and `end()` methods of containers sometimes return iterator and sometimes `const_iterator`, depending

on whether the (implicit) this parameter is marked as immutable with `const` or not. In addition to `*` and `&` on the parameter type, modifiers like `const` are also involved in the resolution of overloads.

One more thing: before you write countless overloads for a function, consider using a function template instead (see [Chapter 23](#)). As of C++20, this is even easier with the abbreviated function templates of the form `void func(auto a)`.

Overloading

I have only briefly touched on the topic of overloading here. I recommend that you experiment a bit and get a feel for it. Especially in conjunction with type conversion and default parameters, function overloading in C++ is a powerful and omnipresent feature. If you master it, you will have understood a lot.

7.9 Default Parameter

In the previous section, I defined a function that adds two values:

```
int add(int n, int m) { return n + m; }
```

If you want to write a function that adds more than just two values, you can now overload it:

```
int add(int n, int m, int o) { return n+m+o; }
int add(int n, int m, int o, int p) { return n+m+o+p; }
int add(int n, int m, int o, int p, int q) { return n+m+o+p+q; }
```

Here you have various versions of `add`, which allow you to sum up to five numbers at once. They all share the same name `add`. But wouldn't it be convenient to write just one function? This becomes even more practical if you need more than five variants.

You can achieve this with the help of *default arguments*. Simply write default values for the parameters, which will be used when you call them with fewer arguments.

```
// https://godbolt.org/z/fjc3rvEr
int add(int n=0, int m=0, int o=0, int p=0, int q=0) {
    return n+m+o+p+q;
}
int main() {
    std::cout << add(1,2,3,4,5) << " ";
    std::cout << add(1,2,3,4) << " "; // like add(1,2,3,4,0)
    std::cout << add(1,2,3) << " ";   // like add(1,2,3,0,0)
    std::cout << add(1,2) << " ";     // like add(1,2,0,0,0)
```

```
std::cout << add(1) << " ";      // like add(1,0,0,0,0)
std::cout << add() << " ";      // like add(0,0,0,0,0)
}
```

Listing 7.10 Default parameters act like multiple overloads.

For example, if you omit the last parameter with `add(1,2,3,4)`, the compiler will automatically insert it for the call. The same applies to further calls: because I have provided all parameters with `=0` as a default value, you can even call `add()` without any arguments, and the compiler will fill `n` to `q` with `0`.

There are a few simple rules for default parameters:

- You can only omit parameters from the end toward the beginning of the parameter list when calling a function. For example, in the previous example, it would be impossible to omit `n` but provide a value for `m` or any other parameter when calling.
- The default value must be a constant (a `constexpr`). A numeric literal is therefore not a problem, and the compiler would also accept `3*4+5`.
- You can also have multiple overloads, all of which take default parameters. In combination, you must ensure that there are no ambiguities.

For example, you can have two `add` overloads for `int` and `double`, each with a set of default parameters:

```
// up to three int parameters:
int add(int n=0, int m=0, int o=0) { return n+m+o; }

// 0 ... 3 doubles:
double add(double a=0., double b=0., double c=0.) { return a+b+c; }
```

If you call `add(1,2)`, the `int` variant will be selected, and for `add(1.25, 3.75)`, the overload for `doubles` will be chosen. In each case, the last parameter will be filled by the compiler with the specified default value.

To avoid conflicts, it should be briefly mentioned that when defining with three `int` arguments, you should not add another one like this:

```
int add(int x); // ⚡ conflict with three-int overload
```

Because if you then call `add(4)`, the compiler will always tell you that it cannot decide between the three-argument and the one-argument overload. In this case, the conflict is still easy to recognize. When you later deal with function templates, it can become more difficult.

7.10 Arbitrary Number of Arguments

The `add` function discussed previously can take up to five arguments, and it would certainly not be difficult for you to extend this to 20 or even 100 default arguments. I don't want to leave the example as is without giving you a glimpse that you can handle an *arbitrary* number of arguments. If you want an `add` function that takes an arbitrary number of `int` values, take a detour via `std::initializer_list`:

```
int add(std::initializer_list<int> ns) {  
    return std::accumulate(begin(ns), end(ns), 0); // or a for loop  
}
```

Now, when calling, you just need to add a few curly brackets around the arguments, but they can be any number. For example:

```
add({1,2,3,4,5,6,7,8,9})
```

7.11 Alternative Notation for Function Declaration

There is another way to write the function header with `auto` and `->`, which is at least equivalent, if not better:

`auto` *FunctionName*(*ParameterType* *ParameterName*,...) *->* *ReturnType*

For [Listing 7.2](#), it would look like this:

```
auto func() -> int;  
auto func() -> std::string;  
auto func() -> void;  
auto func() -> std::pair<int, std::string>;  
auto func() -> vector<int>;  
auto func() -> vector<int>&;  
auto func() -> const vector<int>&;
```

Listing 7.11 Alternative syntax for function declarations with trailing return type.

The advantage is that you can recognize the function name better when the return type becomes more complex. This can happen with templates that have multiple type arguments. For programmers who are used to C, this notation looks very strange, but there is really no reason not to use it—except that it will probably take a long time to catch on, if at all.

From C++14 onward, you can omit the return type in function definitions and let the compiler deduce it instead. The compiler then deduces the type based on the first return statement.

```
// https://godbolt.org/z/6d5fxn9ed
auto maxOf2(int a, int b) {
    return a < b ? b : a; // one return: the compiler determines int
}

auto minOf3(int a, int b, int c) {
    if(a < b) return a < c ? a : c;
    else return b < c ? b : c;
}

auto medianOf3(int a, int b, int c) {
    // more complex, but no problem for the compiler
    return minOf3(maxOf2(a,b), maxOf2(b,c), maxOf2(a,c));
}
```

Listing 7.12 Since C++14, you can let the compiler deduce the return type.

7.12 Specialties

There are still some special cases regarding function definitions that I will briefly mention here so that you can recognize them when you stumble upon them.

For example, you can use `return` to return more than one value. To do this, use the data type `tuple`. I will go into more detail in [Chapter 28, Section 28.1](#). There you will see how to easily bundle multiple values into a single return value using `tuple`—which feels like returning multiple values.

7.12.1 “noexcept”

If you find the `noexcept` keyword after the function declaration, then the author of the function promises that this function will not throw an *exception* (see [Chapter 10](#)). Let's assume the hypothetical class `File`, which has the following function:

```
File openFile(const char* filename) noexcept; // Function does not throw an exception
```

The function `openFile` returns a new `File` object. If anything goes wrong, there are many C++ libraries that throw an exception. This function promises with `noexcept` not to do that; `File` probably has other mechanisms to detect errors.

This information might be useful for users, but it is mainly intended for the compiler. Normally, it has to generate a little bit of code in case an exception¹ *could* be thrown. With this keyword after the function declaration, the compiler can skip that.

¹ The additional effort is so minimal that some even speak of it as *zero-cost*.

7.12.2 Inline Functions

At runtime, for a function call, a stack area is created where the function stores its data. You will also find the return address stored there so that the function knows where it was called from and can return there. In addition, the stack area is used to store local data, local parameters, and the return type. Modern CPUs can handle all of this easily. What hurts, however, is that caches and pipelines need to be cleared for a function call—things that make modern CPUs really fast. This is a considerable effort for a simple function call. If speed is crucial and such a function is called frequently in a loop, then unnecessary computation time is wasted.

For such purposes, you can mark a function with the keyword `inline`:

```
inline int simple_func(double a, double b, double c);
```

With the keyword `inline` before the function, you suggest to the compiler that it should not integrate the function code as a callable function but should insert it in place of the function call.

Suggestion for the Compiler

The term *should* is used here because it is merely a suggestion for the compiler. Whether the compiler integrates the function in place or not is ultimately decided by the compiler itself.

There are other scenarios where the compiler can inline:

- The compiler can decide to inline a function definition from the same source file, even without the `inline` keyword, especially if the function is used frequently.
- If this function definition is also a function template, compilers are even more eager to inline it.
- And at the maximum optimization level, newer compilers even inline arbitrary functions across source file boundaries.

What the compiler actually inlines depends on many factors and is hard to predict.

7.12.3 “constexpr”

A function whose return is not only marked with `const` but also with `constexpr` must have a quite simple function body that the compiler can compute during translation to insert the result. As a rule of thumb, if you only write a `return` statement in the function body, then the chances are good that the use of `constexpr` is allowed:

```
constexpr int doubledUntil100(int value) {  
    return value<=50 ? value*2 : 100;  
}
```

The chances are also good that this will lead to very fast program code. And unlike with `const` or `inline`, the compiler will inform you if it can no longer compute the result at translation time but when the computation would cost runtime. However, whether the compiler actually precomputes the result in the described manner is still up to it.

With `constexpr`, you can define your own literal data types, as you can see in [Chapter 23, Section 23.7](#).

The highest form for the ultimate minimal time consumption when running the program is the `constexpr` modifier for *immediate functions* introduced in C++20. This ensures that the compiler can compute an expression at translation time and insert it at the call site: as always, it doesn't have to, but it is likely. You can learn more about `constexpr` in [Chapter 13, Section 13.10.8](#).

7.12.4 Deleted Functions

If you want to prohibit a specific use of a function, you can write `= delete` instead of the function body:

```
double add(double a, double b) { return a + b; }
double add(int, int) = delete; // prohibit add(3,4)
```

Without the second line with `= delete`, the compiler would automatically convert 3 and 4 to `double`. By making the declaration for `add(int, int)` known to it, it then chooses this one. However, by using `= delete`, you tell the compiler that you explicitly do not want this behavior, and it will issue an error message.

7.12.5 Specialties in Class Methods

For functions that are part of a class, you will encounter a few more things that you should at least see here so that you recognize them when reading. Starting in [Chapter 12](#), I will go into more detail on this:

```
// https://godbolt.org/z/8Gdso36zf
class Widget : public Base {
    explicit Widget(int); // no automatic conversion from int
    ~Widget();           // Destructor with ~ before the name
    virtual void update(); // prefixed with virtual
    void calc1() override; // suffixed with override
    void calc2() final;    // suffixed with final
    void draw() const;     // suffixed with const
    virtual void paint() = 0; // abstract method
};
```

- With `explicit`, you prevent `int` from being automatically converted to `Widget`. This is used before *constructors* (the initializer) and conversion operators.

- The symbol `~`, a tilde, before the name of a function makes it a *destructor*. This is automatically called when the object is removed. The name of the destructor must match the name of the class.
- The `virtual` keyword prepares the method to be overridden by another implementation in a derived class. This is often used for extensibility in logically related object hierarchies.
- `override` is a programming aid that protects against careless mistakes when overriding with `virtual`. It guarantees that you are indeed overriding a method.
- `final` is also intended for `virtual` methods in a derived class and prevents this method from being overridden again in the future.
- Methods marked with `const` promise not to change the state of the object.
- A virtual method that is set to `= 0` instead of an implementation is a *pure virtual function*. This method must be overridden in a derived class for the class to be instantiable at all. In Java and some other languages, this is called an *abstract method*.

Chapter 8

Statements in Detail

Chapter Telegram

- **Empty statement**
The statement consisting of the semicolon ; ; does nothing.
- **Block statement**
A sequence of statements enclosed in curly braces {...}.
- **Declaration statement**
The announcement of new variables, possibly with initialization.
- **Expression statement**
Any expression can also be used as a statement.
- **if statement**
The conditional execution of further statements.
- **while loop**
Repeated execution of statements, tied to a condition at the beginning.
- **do-while loop**
Like a *while* loop, but with the condition at the end.
- **for loop**
Like a *while* loop, but with an additional initialization and progress part.
- **Range-based for loop**
A loop over all elements of a container.
- **switch statement**
Multiple distinctions with *case* in a single statement.
- **break statement**
Exit the current loop or switch.
- **continue statement**
Continue the current loop with the next iteration.
- **return statement**
Exit the current function.
- **goto statement**
Jump to another statement named with a label.
- **try block**
A block of statements where *exceptions* are handled (preview).

In this chapter, you will learn about the different types of statements in detail. I will demonstrate their use mainly through a practical example.

At the beginning, I will revisit some topics from previous chapters and discuss them in more detail. This way, you will encounter some repetition, but with deeper understanding, and you won't have to keep flipping back.

Statement

A statement is a piece of code that does something; for example, it performs an assignment or a loop. Statements (of a thread) are executed sequentially.

This program calculates all prime numbers within a freely determinable range—quite efficiently for the brevity of the program.

```
// https://godbolt.org/z/Y33Tzcf5T
#include <iostream>           // cout
#include <vector>             // container vector
#include <string>             // stoi
int inputUpTo(int argc, const char* argv[]) {
    /* Determine number */
    int upTo = 0;             // introduce new variable
    if(argc<=1) {             // if statement with then and else block
        std::cout << "Up to what number do you want to calculate prime numbers? ";
        if(!(std::cin >> upTo)) { // check return value
            return -1;          // error in user input
        }
    } else {                  // else part of the if statement
        upTo = std::stoi(argv[1]);
    }
    return upTo;              // return input
}
std::vector primes{2};       // new vector<int> with initialization
bool isPrime(int n) {
    /* primes must be sorted in ascending order */
    for(int factor : primes) { // range-based for loop
        if(factor*factor > n)   // too large to be a divisor at all?
            return true;       // ... then terminate inner loop early
        if(n%factor==0)        // is divisor?
            return false;      // ... then exit
    }
    return true;               // no divisor found
}
```

```

void calculatePrimesUpTo(int upTo) {
    /* Primes calculation */
    /* vector must contain {2} at this point */
    for(int n=3; n<upTo; n=n+2) {    // standard for loop
        if(isPrime(n)) {
            primes.push_back(n);    // is prime – mark as divisor and result
        }
    }
}

void outputPrimes() {
    for(int prime : primes) {        // range based, over all elements
        std::cout << prime << " ";
    }
    std::cout << " ";
}

int main(int argc, const char* argv[]) {
    int upTo = inputUpTo(argc, argv); // declares variable
    if(upTo < 2) { return 1; }        // exit main with nonokay value.
    calculatePrimesUpTo(upTo);
    outputPrimes();
    return 0;
}

```

Listing 8.1 Calculates all prime numbers in a user-defined range.

First, look at the rough structure of this program. After that, I will go into detail about the different statements.

In addition to `main()`, I define four helper functions: `inputUpTo()`, `isPrime()`, `calculateUpTo()`, and `outputPrimes()`. The `inputUpTo()` function receives the same parameters as `main()`, to either evaluate the command line or ask for a number on the console, as you saw in [Chapter 4, Listing 4.10](#). `isPrime()` takes the potential prime number as an argument and returns a `bool` value, which indicates whether the argument is prime or not.

Between the functions, the `primes` *global variable* is defined. Because it is global and not within a function or even within a block, all functions can access it.

The variable is of type `vector<int>`: this is a *container* that is particularly suitable for sequentially adding new elements. I use `primes.push_back(n)` to accomplish this. In C++17, it is sufficient to write `vector` as the compiler infers `<int>` from the `2` during initialization.

8.1 The Statement Block

Anywhere you are allowed to write a single statement, you can also group multiple statements within curly braces `{...}`. You can see the schematic flowchart in [Figure 8.1](#). This is also called a *compound statement*. You do *not* need to end this block with a semi-colon; doing so would create an empty statement, which you should avoid (see [Chapter 9, Section 9.2](#)).

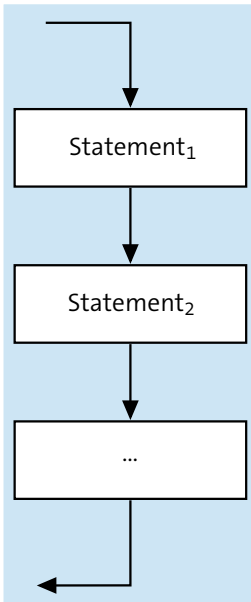


Figure 8.1 Flowchart for the statement block.

In [Listing 8.1](#), I have attached those blocks to most `if` statements and `for` loops. Using `{...}` consistently here prepares for future extensions and improves readability. In `if(factor*factor > n)` and `if(n%factor==0)`, you see `if` without block braces. The same applies to `for`.

```

for(int prime : primes)           // for followed by a single statement
    std::cout << prime << " ";
std::cout << " ";                 // not part of for anymore
  
```

Listing 8.2 This “for” refers to only one statement.

The `for` is only coupled with the first of the two outputs. The second line is output only once. If you wanted to do more things within the loop, then use `{...}`. This way, you get a statement block that counts as a single statement for the `for`.

```

for(int prime : primes){           // Beginning of the block
    std::cout << prime;
    std::cout << " ";
}                                   // End of the block
std::cout << " ";

```

Listing 8.3 A statement block is enclosed in { and }.

Now the separated cout outputs are both executed within the loop.

8.1.1 Standalone Blocks and Variable Scope

However, you don't have to bind blocks to a for or similar; you can introduce them anywhere it makes sense to you instead of a single statement. The following listing is an example of this.

```

if(number > 50) {                  // outer block
{                                  // 1st inner block
    int result = number*number;
    std::cout << "Square: " << result << std::endl;
}
{                                  // 2nd inner block
    int result = number+number;
    std::cout << "Doubled: " << result << std::endl;
}
}

```

Listing 8.4 Where statements are allowed, you can also create a block.

The inner blocks are both executed linked to the if, as they are both within the outer block that belongs to the if.

You can freely choose whether you want to group your statements in this way. This would be useful, for example, if you wanted to introduce the same variable names in both inner blocks. In [Listing 8.4](#), result is introduced as a new int variable in both inner blocks. Variables are only valid within the block in which they were *defined*. Without the division into two blocks, both result definitions would interfere with each other.

```

if(number > 50) {
    int result = number*number;      // Definition of result
    std::cout << "Square: " << result << std::endl;
    int result = number+number;      // ✗ Error: result has already been defined
    std::cout << "Doubled: " << result << std::endl;
}

```

Listing 8.5 Using “result” twice as a new variable in one block is not allowed.

Of course, you could have used the already existing `result` variable here without redeclaring it with `int result = number+number`. However, there are sometimes reasons that you don't want to or can't do that.

You could also have introduced a new variable with `int result2 = ...`. But maybe you didn't want too many variables lying around that you don't actually need anymore. By introducing inner blocks, you made it clear, among other things, where `result` is needed from and to. Later readers then don't have to think about this anymore.

```
if(number > 50) {  
    int result1 = number*number;           // a result  
    std::cout << "Square: " << result << std::endl;  
    int result2 = number+number;           // another result  
    std::cout << "Doubled: " << result << std::endl;  
    int result3 = number+number+number;     // and another result  
    // ... many lines of code in between ...  
    // and here?  
    // ... even more lines of code ...  
}
```

Listing 8.6 Too many variables are also not good.

Someone who wonders at the `// and here?` point whether they can use any of the `result` N variables for their own purposes will only be able to find out with difficulty—and will probably introduce another variable out of despair.

Introducing a new block for each individual variable and indenting left each time fragments the code and unnecessarily lengthens it. Neither is conducive to understanding when reading.

Use common sense when it comes to introducing new standalone blocks.

8.2 The Empty Statement

When you write a semicolon where a statement is expected, that is an empty statement. This statement does *nothing*.

```
int main() { ;;;                               // 3 empty statements  
    int number = 12 ; ;                        // 1 empty statement  
    ; int q = number*number ;                 // 1 empty statement  
    if(q>50) {  
        q = q - 50;  
    } ; ; ;                                   // 1 empty statement  
    std::cout << q << std::endl;  
}
```

Listing 8.7 Empty statements everywhere.

Doing nothing is usually completely uncritical. However, there are cases where an inserted empty statement is dangerous. You already know from [Chapter 4](#) that an `if` is followed by a statement that should be executed if a condition is met. Suppose you insert an empty statement after `if(q>50)` in the preceding example:

```
if(q>50)_; {                // ✗ empty statement with serious consequences
    q = q - 50;
}
```

Then it is the empty statement that gets executed in the case `q>50`. The subtraction `q=q-50` is no longer part of the `if` but is now executed *always*.

Avoid Empty Statements

Even though empty statements are mostly harmless, when they do cause issues, they're a right pain to track down. Develop the habit of not ending your statement block `{...}` with a semicolon.

Be especially vigilant with `if`, `for`, and `while` constructs to avoid inadvertently slipping in empty statements.

8.3 Declaration Statement

We have seen in several places that a new variable was introduced. In [Listing 8.1](#), for example, it was this:

```
int upTo = 0;
```

This introduces a new variable—it *declares* it. First, its *type* is specified—here, `int`—and then the name under which the variable will be used—here, `upTo`. Done. But what about the equals sign `=` and what stands to its right? That is the *initialization*, and it is optional in this form of declaration. I have already discussed the various forms of initialization in [Chapter 4, Section 4.2](#).

So, note the following. With

```
int upTo;
```

is the variable `bis` *declared*. You must not use it as it is now (for calculations and the like) because it does not yet contain a value. More precisely, its value is *undefined*. This means that if you use the variable for calculations, it contains an *arbitrary* value—and in most cases, this will lead to unforeseen behavior in your program. There is more to say about this because many types are also *initialized* with this form of declaration, but not `int`. Read more about this in [Chapter 12](#).

It is also important that variables are only known within the block in which you declare them. Once the program passes the closing brace of the block, they are no longer available. Therefore, this declaration does not overlap with the previous one:

```
int upTo = inputUpTo(argc, argv);
```

This declares a “different” `upTo`, which is only visible in `main()`.

You can also see here that initialization can occur not only with 0 or a constant. Here, a complex function call returns the value with which `upTo` is initialized.

You can also combine declarations separated by commas. You can declare multiple variables like this:

```
int from, to, result;
```

Here, all variables will be of type `int`. If you wanted to initialize all (or some) at the same time, you can do so:

```
int from, to = 0, result;
```

Here you declare three variables, but only `to` is initialized with 0.

8.3.1 Structured Binding

Since C++17, it is also possible to declare multiple elements at once and initialize them all with a single equals sign `=`. To do this, use `auto` before square brackets `[]`, which then contain the variables to be declared.

Three variants are possible: binding a C-array, binding a tuple, and binding a structure.

```
// https://godbolt.org/z/xWbqeeq3n
#include <iostream>
#include <tuple>    // make_tuple
auto mkTpl() {
    return std::make_tuple(2, 'b', 3.14); // tuple<int,char,double>
}
struct Point {
    int x, y;
};
int main() {
    // Structured binding of a C-array
    int odd[5] = { 1,3,7,9,11 };
    auto [ one, two, three, four, five ] = odd;
    // Structured binding of a tuple
    auto [ zwei, be, pi ] = mkTpl();
```

```
// Structured binding of a struct
Point p0{ 10, 15 };
auto [ the_x, the_y ] = p0;
}
```

Listing 8.8 With `auto`, you can also initialize multiple variables at once.

Here, the count must match exactly. So, if `odd` one day contains six elements, binding will fail. The same applies to a tuple that is too large or too small, or to an unsuitable structure.

You don't need to copy out the values; you can also let the declaration be references:

```
// https://godbolt.org/z/3zo8GnbW3
int main() {
    int odd[5] = { 1,3,7,9,11 };
    auto &[ one, two, three, four, five ] = odd;
    auto &[ zwei, be, pi ] = mkTpl();           // ✗ no & binding to temp-values
    Point p0{ 10, 15 };
    auto &[ the_x, the_y ] = p0;
}
```

Here, for example, `three` is a reference to the 7 within `odd`.

This, of course, only works if there is an object that can be referenced. In the case of `mkTpl()`, this is not the case, as the function returns a *temp-value* that is not yet stored anywhere.

8.4 The Expression Statement

You can also use any *expression* as a statement. Although it makes little sense to use `3+4`; as a statement, for example, assignments are expressions and are used as statements very often.

Assuming you have previously declared `int upTo`;, the following listing gives examples of expressions used as statements.

```
upTo = 99;                // an assignment is an expression
upTo = upTo * 2 + 1;      // combining an assignment with a calculation
calculatePrimesUpTo(upTo); // a function call is an expression
std::cout << "Friedrich III"; // the output operator
```

Listing 8.9 These are all expressions used as statements.

As I mentioned earlier, every expression has a value and a type. If an expression is used as a statement, the type disappears and is discarded by the compiler.

Control Structures

The `if`, `while`, `do`, `for`, and `switch` statements are *control structures* and allow you to program branches and loops. With `break` and `continue`, you influence the flow of control structures. `return` also belongs to this family in the broadest sense. I would prefer not to mention `goto` (to find out why, see [Section 8.14](#)).

8.5 The “if” Statement

With the `if` statement, you program the *conditional execution* of statements. The simplest form executes the statement assigned to it only if a specified condition is met:

```
if( Condition ) Statement
```

You already know a bit about the statement part: here you should use a block with `{...}` so that you can group multiple statements and link them to the `if`. This is called the *then branch* of the `if` statement.

The *condition*, however, is still somewhat ominous. This must be an expression that evaluates to a truth value—that is, *true* or *false*. This applies to a specific group of expressions—namely, those that have the type `bool`.

More correctly, it must be possible to *convert* the expression to `bool`. This is a subtle but important difference that I explained in [Chapter 4, Section 4.4](#). This simply means that you can also use a number like 0 or 42 and many other things directly as a condition in `if`. If you know how to handle it, `if(x<30)` and `if(cin>>val)` will come easily to you.

Here are a few conditions that can only be *true* or *false*:

- **`number > 100`**

A simple arithmetic comparison that checks if `number` is greater than 100. You can also test for less than `<`, greater than or equal to `>=`, less than or equal to `<=`, equal to `==`, and not equal to `!=`.

- **`number > 100 && number < 200`**

With the *Boolean AND* `&&`, you check if both expressions are true.

- **`x < 0 || y < 0`**

The *Boolean OR* `||` fits when at least one condition should be met.

- **`x == y+1`**

You can perform arithmetic calculations before a comparison.

- **`(x > 0 || y > 0) && (maxx < 100 || maxy < 100)`**

Parenthesize complex expressions if necessary.

- **`!(x < y)`**

With the operator `!` for *not*, you negate a Boolean expression; *true* becomes *false* and vice versa.

With this repertoire, you are already well-equipped for the `if` condition.

There is another useful form of the `if` statement, which you can also see in [Figure 8.2](#):

```
if( Condition ) Statement1 else Statement2
```

This implements: “Execute either *Statement1* or *Statement2*.” The *Statement2* is called the *else branch* of the `if` statement. Thus,

```
if(x > 1) {
    std::cout << "x is greater than 1" << std::endl;
}
if(!(x > 1)) {           // test alternative
    std::cout << "x is not greater than 1" << std::endl;
}
```

is almost the same as

```
if(x > 1) {
    std::cout << "x is greater than 1" << std::endl;
} else {                // with else you save a comparison
    std::cout << "x is not greater than 1" << std::endl;
}
```

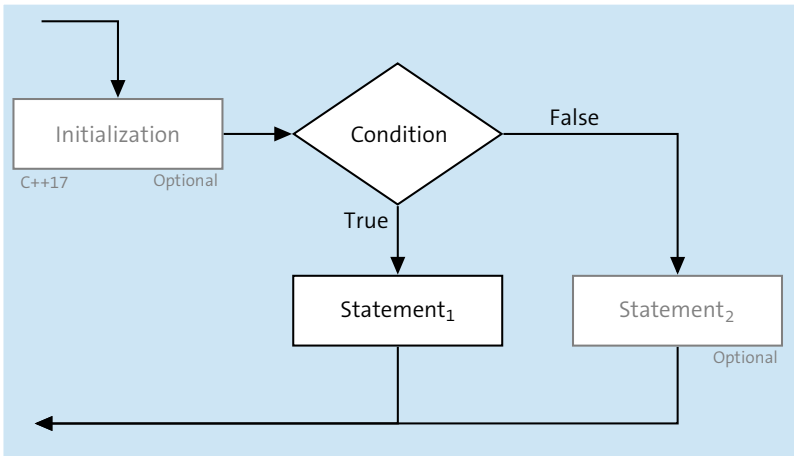


Figure 8.2 Flowchart of the “if” statement with and without the “else” branch.

Why only almost? Because you don't have to execute `x > 1` here twice—and above all, you don't have to write it twice. We have already used blocks `{...}` here instead of a single statement. You could also have written the following:

```
if(x > 1)
    std::cout << "x is greater than 1" << std::endl;
else
    // a block or a single statement follows else
    std::cout << "x is not greater than 1" << std::endl;
```

But because `else` branches make the code more complex, blocks are really recommended here.

Now you actually know everything about `if` that you can know. What remains is to combine the knowledge: An `if` is a statement; the `if` condition, then branch, and optional `else` branch all together is a statement. And both then and `else` each get a statement associated. You can combine this into complex nestings. It is common to use another `if` in the `else` branch to sequentially query different possibilities:

```
if(number < 0) {
    std::cout << "The number is negative.\n";
} else if(number == 0) { // else and if follow directly one after another
    std::cout << "The number is zero.\n";
} else if(number > 1000) { // also possible multiple times
    std::cout << "The number is too large.\n";
} else { /* x > 0 but x <= 1000 */
    std::cout << "The number is positive.\n";
}
```

Here it is common to save yourself from wrapping every `if` statement in a block with only one statement. However, if you once make it clear to yourself where you would place *all* block braces, then you learn something about statements. Completely with all `{...}` around each single statement, the example would look like this:

```
if(number < 0) {
    std::cout << "The number is negative.\n";
} else {
    if(number == 0) {
        std::cout << "The number is zero.\n";
    } else {
        if(number > 1000) {
            std::cout << "The number is too large.\n";
        } else { /* x > 0 but x <= 1000 */
            std::cout << "The number is positive.\n";
        }
    }
}
```

Here it becomes clear which `else` exactly belongs to which `if`. But the ever-deeper indentation comes at the expense of readability. For standard `if-else` cascades, it is better not to wrap the `else` branch in its own block.

8.5.1 “if” with Initializer

Since C++17, you can place an (initialization) expression before the actual condition:

if(Initialization ; Condition) Statement

Typically, you declare a variable in *Initialization*, but you can also write any expression here. A declared variable is valid within the *if* statement, as this is equivalent to the following:

{ Initialization ; if(Condition) Statement }

In a program, it looks like the following listing.

```
// https://godbolt.org/z/d6PscKx4M
#include <map>
#include <string>
#include <algorithm> // any_of
#include <iostream> // cerr
std::map<int, std::string> m;
int main() {
    if(auto it = m.find(10); it != m.end()) { return it->second.size(); }
    if(char buf[10]={0}; std::fgets(buf, 10, stdin)) { m[0] += buf; }
    std::string s;
    if(auto keywords = {"if", "for", "while"};
        std::any_of(keywords.begin(), keywords.end(),
            [&s](const char* kw) { return s == kw; })) {
        std::cerr << "Error ";
    }
}
```

Listing 8.10 These *if* statements contain initializers.

The *it*, *buf*, and *keywords* variables are only valid within the *if* statement. Of course, you can also use an *else* branch. The initialization is also valid within it.

8.5.2 Compile-Time “if”

If you don't want a piece of code to be included in your compiled program under certain circumstances, you can use `constexpr if` since C++17:

if constexpr(Condition) Statement

The compiler then evaluates the condition at compile time and has the option to exclude an entire piece of code from the final program. This really makes sense for debug constants or generic functions.

```
// https://godbolt.org/z/Ycse4EWav
#include <iostream> // cout

template<typename T>
void memory(T x) {
    if constexpr(sizeof(T) > 4) {
        std::cout << "Requires a lot of memory: " << x << " ";
    }
}

constexpr auto DEBUG = true;
int main() {
    if constexpr(DEBUG) {
        std::cout << "Debug is on. ";
    }
    memory<long long>(44LL);
}
```

Listing 8.11 The compiler can evaluate the condition at compile time.

Such things were previously only possible with macros or clever template programming.

You can certainly add an `else` branch and use it together with an initialization expression.

Starting from C++23, there is also an `if constexpr`, which determines if the calling context is `constexpr`; see [Chapter 13, Section 13.10.9](#).

8.6 The “while” Loop

The `while` loop executes a statement so long as a specified condition is met. The process is shown in [Figure 8.3](#). You write it like this:

`while( Condition ) Statement`

For *Condition* and *Statement*, everything I mentioned about `if` applies. Therefore, the following example should pose no difficulty for you.

```
// https://godbolt.org/z/h466fYq4W
#include <iostream>
int main() {
    /* Sum up from 1 to 100 */
    int sum = 0;
    int number = 1;
    while(number <= 100)    // condition
```

```

{                                     // block that is repeatedly executed
    sum += number;                   // for the result
    number += 1;                     // next number
}                                    // end of the repeated block
std::cout << sum << std::endl;
}

```

Listing 8.12 The loop runs 100 times.

Before the loop block is executed, the condition is checked first. Therefore, this loop will run for every `number` between 1 and 100 inclusive—exactly 100 times. And each individual number will be added to `sum`. At the end, `sum` contains the result of adding all numbers from 1 to 100—the task that the mathematician Gauss is said to have solved in his head within seconds in elementary school because he supposedly realized he only needed to calculate $(100 \cdot 101) / 2$. Well, with the help of the computer, you are even faster than Gauss.

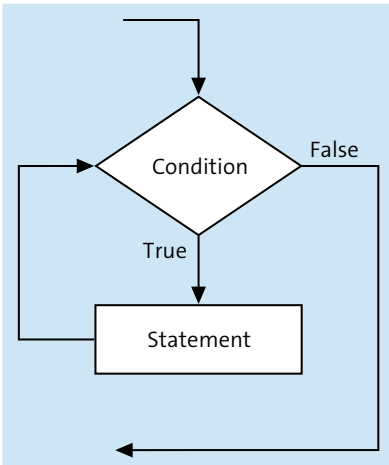


Figure 8.3 Flowchart of the “while” loop.

When you come up with such loops, it is very important to ensure that the condition eventually no longer holds true and the loop then terminates. Here, `number` is a simple counting variable and is incremented within the block. Common mistakes include forgetting to increment or setting the condition incorrectly. Can you see what happens if you forget `number = number + 1`?

```

while(number <= 100) {               // ✗ number is checked but never incremented
    sum += number;
};

```

This loop will run forever because `number` is never incremented, so the condition `number <= 100` will always be true. The program will never reach the line for the output, and you

will have to terminate it from the outside. If you don't yet know how to do that, now is the time to find out. In an IDE, you will find a button or menu item for this. In the console, `Ctrl+C` usually saves you.

It may also be that the condition is not met right from the start. In that case, the block of statements is not executed at all.

Loop Design

Have you noticed that `number` is 101 when the loop ends, not 100? It has to be this way as otherwise the condition would still be met, and the loop would run one more time. Therefore, the *postcondition* of the loop is that after its complete run-through, `number > 100`.

Loops are a challenging aspect of programming and a common source of errors. The mistake is not always as easy to find as forgetting to increment. To work safely, make it clear to yourself what roles the involved variables have in the loop: Is the variable the counter? Is it mutable? And what is its relationship to the condition?

8.7 The “do-while” Loop

This much rarer variant of the `while` loop has the condition at the end:

do *Statement* while(*Condition*)

Here, the *Statement* is executed first and then the *Condition* is checked, as you can see in the flowchart in [Figure 8.4](#). The check occurs after each iteration. This means that the statement is executed at least once.

```
// https://godbolt.org/z/6hh5KhExM
#include <iostream>           // cin
#include <string>
int main() {
    std::string line;
    do {                      // execute getline at least once
        std::getline(std::cin, line);
        if(!std::cin) break;  // error or end of file
    } while(line != "quit");  // end on specific input
}
```

Listing 8.13 The body of a “do-while” loop is executed at least once.

You should not use this variant of the `while` loop with `do` very often as it is quite uncommon. In addition, the condition in a loop is so important that having it at the end can hinder understanding when reading.

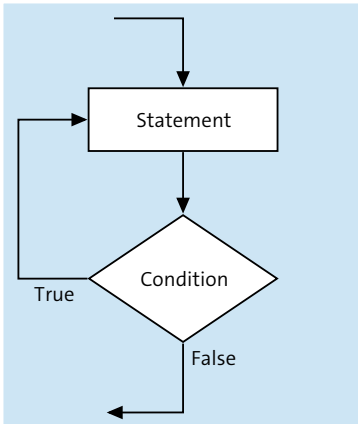


Figure 8.4 Flowchart of the “do-while” loop.

8.8 The “for” Loop

You have seen in the `while` loop that you often initialize a counter variable for the loop and then increment it at the end of the loop. Because this is so common, it is packed together and thus simplified in the `for` loop:

`for( Initialization ; Condition ; Update ) Statement`

Statement and *Initialization* are statements, while *Condition* and *Update* are expressions.

It is common to define a loop variable in the initialization, modify it in the update, and check it in the condition.

A `for` loop corresponds to this well-organized `while` loop:

```
{ Initialization ; while( Condition ) { Statement ; Update ; } }
```

In [Figure 8.5](#), you can see the process schematically.

I will rewrite [Listing 8.12](#) into a `for` loop in the following listing.

```
// https://godbolt.org/z/7foYcMMaz
#include <iostream>
int main() {
    /* Sum up from 1 to 100 */
    int sum = 0;
    for(int number=1; number <= 100; number+=1) { // compact form
        sum += number;                          // for the result
    }
    std::cout << sum << std::endl;
}
```

Listing 8.14 Sum with a “for” loop.

Here is the initialization `int number=1`. Before the statements in the loop are executed, the condition `number <= 100` is checked. Then follows the execution of the loop, here `sum += number;`. At the end of the block, the update is executed with `number += 1`.

Then the condition is checked again. If it is met, the next loop iteration follows; if not, the loop ends.

Even with `for`, `number` ends up at 101. As I have written it, I cannot output it because the `number` variable only exists within the loop. However, you can declare it before the `for` and leave the initialization part empty.

```
// https://godbolt.org/z/185ha3q3f
#include <iostream>
int main() {
    /* Sum up from 1 to 100 */
    int sum = 0;
    int number = 1;                // Initialization before the loop
    for( ; number <= 100; number=number+1) { // empty initialization
        sum = sum + number;
    }
    std::cout << number << std::endl;    // number now also exists outside
}
```

Listing 8.15 “for” loop with empty initialization part.

Here you can also access `number` outside the loop and thus see how 101 is printed.

It is also possible to leave the update part empty. Then you will have to take care of the update within the loop block. But this is rather rare, because then you almost have a `while` loop. In extreme—and particularly rare—cases, you can also omit the condition. Then you must be able to end the loop in some other way. How to do this without `Ctrl+C` is something you will learn when I explain `break` in this chapter.

A frequently used way to execute something “forever” is to omit all three parts.

```
int main() {
    for(;;) { // no init, no condition, no update – so forever
        /* ... User input */
        /* ... if user chooses Quit, end program */
        /* ... otherwise, perform calculation and output */
    }
}
```

Listing 8.16 Loops without conditions must be terminated in some other way.

Of course, a lot of program code is missing here. But using a loop that runs forever as a last resort should make the use case clear to you.

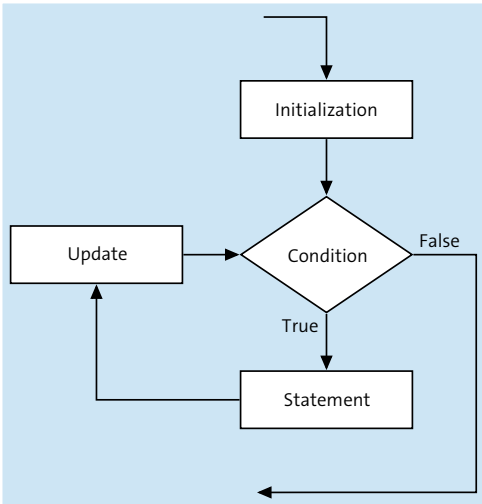


Figure 8.5 Flowchart of a “for” loop.

8.9 The Range-Based “for” Loop

When I explain the `for` loop, I must also cover its new variant, the *range-based for loop* (or *ranged for*). Especially with *standard containers*, it is very useful.

To examine the contents of the `prims` container, I use the `for` loop with the colon notation `::`. I refer to the following excerpt from [Listing 8.1](#).

```
// Excerpt
std::vector<int> primes{2};    // vector is a container, ready for a ranged for
void isPrime(int n) {
    /* primes must be sorted in ascending order */
    for(int factor : prims) { // range-based for loop
        if(factor*factor > n)
            return true;
        if(n%factor==0)
            return false;
    }
    return true;
}

void outputPrimes() {
    for(int i=1; int prime : prims) { // range-based for loop
        std::cout << i++ << ". Prime number: " << prime << " ";
    }
    std::cout << " ";
}
```

Listing 8.17 You can recognize the range-based “for” loop by the colon.

`primes` collects the overall result so that it appears on the screen in `outputPrimes`. On the other hand, it is used in `isPrime` to perform only the necessary divisibility tests for the previous prime numbers.

You can read `for(int factor : primes)` this way: Select an element from the `primes` container one by one, assign it to `factor`, execute the loop statement, and repeat this process for all elements of the container. That is:

for(Destination : Container) Statement

That already allows you quite a lot. You can even equip your own data types to be used in these `for` loops like a container. All you need to do is appropriately define `begin` and `end` methods.

When outputting with `int prime : primes`, it is the same, but here we also have an initialization part with `int i=1`; before it. This has been possible since C++20. Before that, you had to define such a counter variable before the loop.

Normally, you apply the range-based `for` to containers, which you will get to know in [Chapter 24](#). If you write container-like classes yourself, you can make them fit the range-based `for`.

8.10 The “switch” Statement

You have already seen that you can use `if-else` cascades to check multiple cases in succession. The `switch` statement is a specialized form of the case where you always test the same expression, and the expression has an `int`-like type—or is an `enum`, as you will see in [Chapter 16, Section 16.12.2](#):

switch(Expression) { Cases }

Since C++17, it works analogously to `if` as well:

switch(Initialization ; Expression) { Cases }

Where *Cases* consist of a sequence of cases of the following form:

case Constant: Statements

From the list of *Cases*, those *Statements* are executed whose *Constant* matches the result of the *Expression*. There are still some things to consider or that are worth knowing:

- *Cases* must not contain duplicate *Constants*, but they also do not need to follow a specific order.
- Instead of `case`, you can also use `default` once to catch everything else. Normally, this is the last or first case, but it doesn't have to be.
- You can also write multiple `case` instructions directly in a row without statements in between if you want to combine multiple cases.

- Statements of a case should always end with `break`. Exceptions to this rule are very rare and should be clearly highlighted with a comment. If you forget this `break`, the statements of the next case will be executed, regardless of what its constant is. This is called *fall-through*.

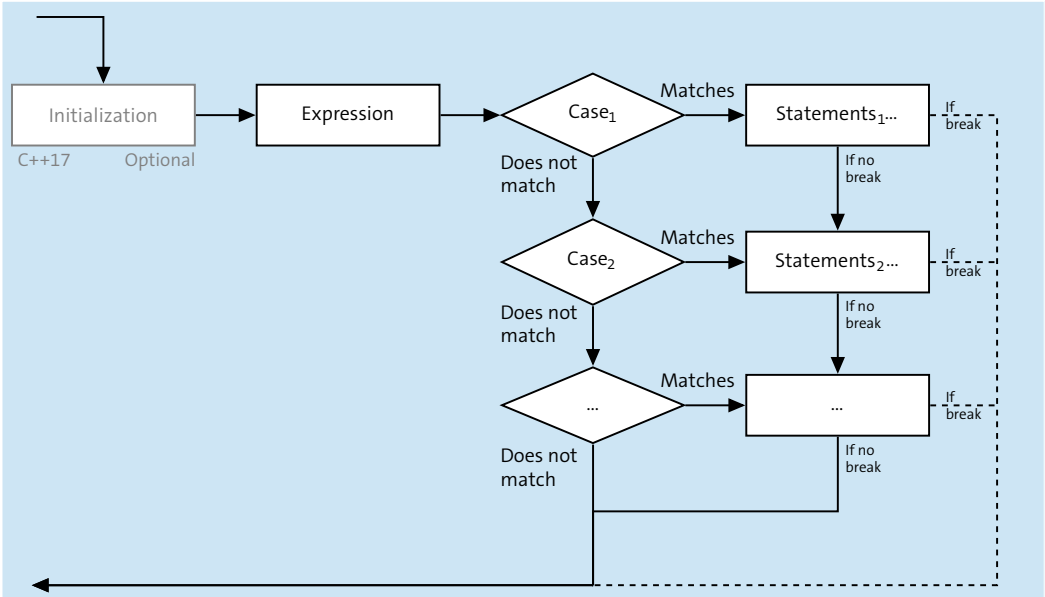


Figure 8.6 Flowchart of a “switch” statement.

In [Figure 8.6](#), you can see the flowchart of the `switch` construct. Although the `break` statements do not strictly belong to the `switch` flowchart, I have included them here as an option because they are so common and important.

The very simple calculator from [Listing 8.18](#) places digits in a stack. An arithmetic operation takes the top elements of the stack and places the result back on top.

```
// https://godbolt.org/z/Y4W1oW8b
#include <string>
#include <vector>
#include <iostream>          // cout

void calculator(std::ostream& out, std::string input) {
    std::vector<int> stack {};
    for(char c : input) {
        if(c>='0' && c<='9') {
            stack.push_back( c-'0' ); // numeric value of the character
            continue;                // next loop iteration
        }
    }
}
```

```
int top = 0;
int second = 0;
switch(c) {           // condition on character
case '+':
    top = stack.back(); stack.pop_back();
    second = stack.back(); stack.pop_back();
    stack.push_back(second + top);
    break;
case '-':
    top = stack.back(); stack.pop_back();
    second = stack.back(); stack.pop_back();
    stack.push_back(second - top);
    break;
case '*':
    top = stack.back(); stack.pop_back();
    second = stack.back(); stack.pop_back();
    stack.push_back(second * top);
    break;
case '=':
    for(int elem : stack) { out << elem; }
    out << " ";
    break;
case ' ':
    break;
default:
    out << " '" << c << "' I don't understand. ";
} /* switch */
} /* for c */
}

int main(int argc, const char* argv[]) {
    if(argc > 1) {
        calculator(std::cout, argv[1]);
    } else {
        // 3 + 4 * 5 + 6 with multiplication before addition results in 29
        calculator(std::cout, "3+4*5+6=");
    }
}
```

Listing 8.18 Each “case” covers one scenario, and nowhere was the “break” forgotten.

Much more relevant, however, is `switch(c)`. As a `char`, you can use `c` here, just like `int` and its relatives. However, you must not write a `double` or `string` here. It must be an “enumerable” type.

The case cases follow one after another. Depending on the value of `c`, these jump labels are directly accessed, and the subsequent statements are executed. Each block is properly concluded with a `break`; to prevent accidentally executing another calculation.

A small comment on the calculation `c - '0'`: Here, I subtract the character value `'0'` from the entered `c` to get the numeric value of the character. Mind you, this is not the `int` number 0, but the `char` character `'0'`. The result is an `int` number between 0 and 9. This works partly because `char` behaves very much like an integer type and partly because the characters `'0'` to `'9'` have consecutive values as `int`. The `'0'` usually has the number 48 and the `'4'` (for example) the value 52. Thus, `'4' - '0'` is the same as `52 - 48`, which is the `int` 4. I then put this numeric value onto the stack.

It Can Be Better; It Can Be Dynamic

The `char`-based `switch` is quite nice as a small example, but it is very static and can become unwieldy when you want to extend it. However, if you want to see a version that is better suited as a basis for an elaborate calculator, flip far ahead to [Chapter 28, Listing 28.43](#). There, it is a dynamic map that takes function objects for the operations.

If you're wondering why the `calculator` function has an `out` parameter: It's always good to never use the global variables `cout`, `cerr`, or `cin` inside of functions. This makes such functions more testable. You should only access these global variables in `main()`. Thus, `cout` ends up here as a parameter of `calculator()` in the function. You can see such a test in [Chapter 12, Listing 12.8](#).

Don't Forget “break”

Never forget the `break` that concludes a case block. In 99.99% of cases, you need it to prevent falling through to the next case block. Forgetting it is a common oversight.

If you ever intentionally program a fall-through, write an unmistakable comment at that point. Otherwise, anyone else will think at first glance that you made a mistake and forgot the `break`;

Cases where a case block is not concluded with a `break` are rare. Often, a series of separate `if` statements will suffice. [Listing 8.19](#) will help you understand the principle of fall-through.

```
// https://godbolt.org/z/jaMTcshEf
#include <iostream>
#include <string>

using std::string; using std::cout;

void guessMonth(unsigned whatDayIsItToday) {
```



```
switch(whatDayIsItToday) {  
    /* missing break statements: fall-through intended */  
    default:  
        if(whatDayIsItToday>31) {  
            cout << "You are cheating";  
            break;  
        }  
    case 28:  
    case 29:  
        cout << "Feb ";  
    case 30:  
        cout << "Apr Jun Sep Nov ";  
    case 31:  
        cout << "Jan Mar May Jul Aug Oct Dec ";  
}  
  
cout << ". ";  
}  
int main() {  
    guessMonth(31); // if today were the 31st?  
    // Output: Jan Mar May Jul Aug Oct Dec .  
    guessMonth(30); // what if it were the 30th?  
    // Output: Apr Jun Sep Nov Jan Mar May Jul Aug Oct Dec .  
    guessMonth(4);  
    // Output: Feb Apr Jun Sep Nov Jan Mar May Jul Aug Oct Dec .  
    guessMonth(77);  
    // Output: You are cheating.  
}
```

Listing 8.19 Very rarely are there “case” blocks without “break”—and even more rarely meaningful ones.

For `guessMonth(30)`, the switch directly jumps to the label `case 30:` and skips the cases `default`, `28`, and `29`. For `30`, “Apr Jun Sep Nov ” is then output. Because there is no `break;` at the end of the block for `case 30`, the statements of the next block are executed, so “Jan Mar May Jul Aug Oct Dec ” is output.

For `guessMonth(4)`, no case matches, so the `default:` block is executed. For `4`, the test in the `if` will result in `false`. The end of the block is reached, but due to the missing `break;`, the following case blocks are executed.

Only in the case of `guessMonth(77)` is a `break` reached. Although the `default` block is also executed, the `if` condition is true this time, and the `break;` in the `then` branch is thus reached. With the `break`, the entire switch is now exited, and it continues directly with `cout << ". ";`.

8.11 The “break” Statement

The `break` keyword allows you to exit the current loop—the innermost one, from the perspective of `break`. The program continues with the statement following the loop. You can use this in all `for` and `while` loops, including the range-based `for` and the `do-while`. In this sense, `switch` is considered a loop.

```
// https://godbolt.org/z/75h9Y95vf
#include <iostream>                                // cout

int main() {
    for(int x=1; x<20; x+=1) {                    // outer for-loop
        for(int y=1; y<20; y+=1) {                // inner for-loop
            int prod = x*y;
            if(prod>=100) {
                break;                            // exit inner y-loop
            }
            std::cout << prod << " ";
        } /* end for y */
        // destination of break
    } /* end for x */                             // first actual line after break
    std::cout << " ";
}
```

Listing 8.20 With `break`, you terminate a loop prematurely.

Let's assume the task is to output only the two-digit results of the products of `x` and `y` between 1 and 20. Once `y` is large enough that the product is 100 or greater, then an even larger `y` will not yield a smaller product. In that case, we can terminate the `y` loop prematurely using `break`.

The rest of the `y` loop is skipped; in particular, the output of `prod` is not executed. Rather, execution continues *with the first statement following the inner loop*. Well, this is a special case: here, at `/* end for y */`, the end of the `x` loop is reached, where no more statements are present. Therefore, the next thing to be executed is the update expression of the `x` loop—that is, `x+=1`.

Note that only the current *innermost* loop will be exited by the `break`, meaning that the `x` loop will not be exited. If you had placed a `break` after `/* end for y */`, then the `x` loop would be exited and continued at `cout << " "`.

With `break`, you only exit loops like `for` and `while`, as well as the `switch` (which is not a loop). An `if` does not count as a loop when searching for the current “innermost” loop.

8.12 The “continue” Statement

The `continue` statement has a similar role as `break`. However, the current loop is not *aborted*; only the rest of the current loop iteration is skipped, and the loop continues with the next *iteration*. In the `for` and `while` loops, the loop header is executed next. That means, in `for`, that the update and condition are executed, and in `while` (and `do-while`), that the condition is checked. If the condition still holds, the loop body instructions are executed again.

You saw an example in [Listing 8.18](#). When the program execution hits `continue`, it immediately jumps back to `for` and fetches the next `char c` from input.

In a `switch`, a `continue` has no meaning like `break`. It thus affects an enclosing loop.

8.13 The “return” Statement

You have already seen `return` as a statement to exit the program from `main`. Besides `main`, you can use it to exit any function you write.

```
// https://godbolt.org/z/5ddxa9vj
#include <iostream>                                // cout
int min3(int x, int y, int z) {                    // function returns an int
    if(x<y) {
        if(x<z) return x;
        else return z;
    } else if(y<z) {
        return y;
    }
    else return z;
}
void printMin(int x, int y, int z) {               // function returns nothing
    if(x<0 || y<0 || z<0) {
        std::cout << "Please only numbers greater than 0\n";
        return;
    }
    std::cout << min3(x,y,z) << "\n";
}
// no return here
int main() {
    printMin(3, -4, 8);
    printMin(6, 77, 4);
    return;                                        // special return in main
}
```

Listing 8.21 With “`return`”, the current function is exited. If necessary, provide a value for the return.

In this example, you see the two self-written functions `min3` and `printMin` (as well as the familiar `main`). `min3` is supposed to return an `int` as a result, while `printMin` is specified with `void`, indicating that *no* return value is expected.

Because `min3` is supposed to return an `int`, every `return` in it must be followed by an expression of type `int` (or one that can be converted to it). This is the case for all `return` lines in the function, as the variables are of type `int`.

For `printMin`, however, no return value is expected, so the `return;` must not receive an expression. Note that execution can “fall out” of the function without encountering a `return`; this is allowed in `void` functions.

You already know that `return` has a special role in `main`: although `main` has the `return` type `int`, a `return` here—and only here—does not need to receive a return expression, as shown in the final `return` of `main`. It is then assumed to be 0.

C++20: Coroutine Statements

With C++20, coroutines introduce a new feature to the C++ language. They enable generators and simpler asynchronous programming. I won't go into detail here, but I would like to mention that you might encounter two new keywords where previously only `return` could be used:

- `co_return Expression ;`
- `co_yield Expression ;`

You need this and `co_await` to turn a function into a *coroutine*—a function that you can resume at the same point after leaving.

8.14 The “goto” Statement

The question of whether to use `goto` is complicated. Let me summarize it this way: it is difficult to use `goto` correctly, and it is easy for something bad to come out of it.

My tips for you when programming in C++: Avoid using `goto`. Always check all alternatives first when you think about using it. Only when you are familiar with all other ways to structure your program, and `goto` remains the best option, can it sometimes be used effectively.

If you have not yet mastered those alternatives and therefore use `goto`, you will make your code unclear and unmaintainable in the long run.

When a `goto` might be better, you can read, for example, a key work from Donald Knuth¹ (an excellent paper, of which I suggest you read at least the introduction). Until

¹ *Structured Programming with go to Statements*, D. E. Knuth [1974]

then, the rule is to avoid using `goto`. And so you know what to avoid, the syntax of a `goto` statement looks like this:

`goto Label`

Where a label is defined somewhere else:

Label : Statement

When the program execution encounters the `goto`, it immediately jumps to the statement labeled as such.

It is difficult to find a somewhat meaningful example for `goto`. I apologize that the following example is even more contrived than usual.

```
// https://godbolt.org/z/ohd5Pr8fb
#include <iostream>
int main() {
    int idx = 4;
    goto more;                // jump to label more
print:                        // label for the next statement
    std::cout << idx << std::endl;
    idx = idx * 2;
more:
    idx = idx + 3;
    if(idx < 20)
        goto print;          // goto can also be used in an if statement
end:                          // this is a label, but it is not used
    return 0;
}
```

Listing 8.22 Avoid `goto` statements.

The danger is that when using `goto`, you jump wildly back and forth. Or can you quickly determine which numbers [Listing 8.22](#) outputs? The answer is 7 and 17, but figuring that out is tedious.

With the alternatives described in this chapter, you have enough tools at your disposal to manage without `goto`.

In [Listing 8.1](#), I actually introduced the `isPrime()` function only because I couldn't jump behind the `push_back` with a `break` or `continue`; I would have had to introduce and manage an extra `bool` variable. Maybe such a case is a justifiable use of `goto`? But please only use it in code that will never be extended and where there is no risk of the `goto` moving far from the label.

```
// https://godbolt.org/z/fnsWhn5T3
// #includes, inputUpTo(), outputPrimes() and main() as before
std::vector<int> primes{2};
```

```

void calculatePrimesUpTo(int upTo) {
    /* Primes calculation */
    /* vector must contain {2} at this point */
    for(int n=3; n<upTo; n=n+2) {
        for(int factor: primes) {
            if(factor*factor > n)
                goto prime;      // using goto, because a break ...
            if(n%factor==0)
                goto notPrime;   // ... doesn't work over two loops.
        }
        prime: ;                  // Target of the jump before push_back
        primes.push_back(n);      // n is prime! Remember as divisor and result
        notPrime: ;              // Target of the jump after push_back
    }
}

```

Listing 8.23 One less function, but one label and two more “goto” statements.

8.15 The “try-catch” Block and “throw”

Exceptions are an important but advanced topic. Because the try with the corresponding catch is the only remaining statement, I will still give you a preview. I will go into detail in [Chapter 10](#).

With a try block, you enclose program code where you expect certain errors that you want to handle. You handle these errors in the subsequent catch.

```

// https://godbolt.org/z/jhPaEjh1r
#include <iostream>
int main() {
    try {                                // beginning of the try block
        for(int n=1; ; n=n*2) {
            if (n > std::numeric_limits<int>::max()/2) { // check for coming overflow
                throw "There would be an overflow";
            }
        }
    }                                    // End of the try block
    catch(const char *error) {           // if this error occurs, ...
        std::cout << "Error: " << error << " "; // ... handle it this way
    }
}

```

Listing 8.24 As a preview, here is your first exception handling with “try” and “catch”.

A `try` is followed by a statement block `{...}`. This is followed by the handling of errors that occur within the block. This error handling is initiated with `catch`, followed by the type of error to be caught in parentheses. After the parentheses, there is a block `{...}` with the instructions for the case in which an error of this type occurs.

If an error of type `const char*` is triggered somewhere between `try {` and the corresponding brace `}`, it will be “caught” in `catch(const char *error)` and handled in the associated statements. After that, the program continues with the next statement after the entire `try` block, including all associated `catch` blocks.

In this example, I trigger such an error with `throw` because the `if` checks if the next `n=n*2` would overflow. Immediately, the normal execution of statements is interrupted and instead continues with the corresponding `catch`. This is just a simplified explanation; it gets really exciting when you trigger such errors while deeply nested in function calls. That's when these exception-handling mechanisms show their strengths.

You can specify multiple `catch` blocks for one `try`. The first matching one will be executed when an error occurs.

A word about how the error was actually triggered in this program: Could we just check `if(n<0)` when the `int` overflowed and wrapped around? Well, there is a small *yes* and a big *no*: while on most machines this indeed might happen and work, the C++ standard considers integer *overflow* undefined behavior and is therefore to be avoided (see [Chapter 4, Section 4.4.3](#)).

8.16 Summary

You have now learned all the types of statements that C++ offers. I will summarize them again in groups:

- **Simple statements**
Statement block `{...}`, declaration, expression—and empty statement
- **Branches**
`if` with and without `else`, `switch` with `case`
- **Loops**
`for`, range-based `for`, `while`, and `do-while`
- **Jump statements**
`break`, `continue`, and `return`—and, if necessary, also `goto`
- **Exception handling**
`try-catch` block with `throw` statement

Chapter 9

Expressions in Detail

Chapter Telegram

- **Result**

The result of an expression can be reused.

- **Side effect**

A side effect occurs when an expression not only provides a result but also makes further changes to the state of the computer.

- **Overloading**

Multiple functions with the same name, which the compiler distinguishes based on different types of parameters.

- **Literal**

A concrete value directly written in the source code.

- **Parentheses**

Use them to group sub-expressions within expressions.

- **Function call**

The function name is followed by parentheses containing the list of arguments.

- **Index access**

When accessing an index, you use square brackets [...] to access an element among several similar ones.

- **Assignment**

An expression with the left side as the target, the equals sign =, and the right side as the source.

- **Operator**

A function-like construct where the function name is often a symbol and sometimes stands between two arguments.

- **Type conversion (cast)**

Forcing the type of an expression to another type.

With our discussion of statements, you've deepened your knowledge of a very important basic element. Statements usually consist of *expressions*, which we have only briefly discussed so far and which I will now examine in detail. And last but not least are types, which form the most complex topic of these three.

Statements, expressions, and types are closely related. It is difficult to explain one without the other. That's why I chose the approach of confronting you with all three one after the other and going into more detail each time. With this chapter, you will reach the penultimate level of detail for expressions together with types. The final level is programming practice.

9.1 Calculations and Side Effects

According to the C++ standard, an *expression* is “a sequence of operators and operands that specifies a computation.” (The exact definition is, of course, also given in the form of formal computer science grammar, but that is of little use for learning, and more as a reference.)

This *computation* is what programming is really about. And there I have two larger areas to distinguish:

- **Result**

There is a direct result of a calculation, like `sin(alpha)` or `count*4`. This can be assigned or further used, for example in `x = sin(alpha)` and `print(count*4)`. For C++, the *type* of this result is also important, as it determines how to proceed; see [Listing 9.1](#).

- **Side effect**

Instead of or in addition to the result, more happens; the further “state” of the computer changes. For example, `std::cout << x` results in an output as a side effect. This expression does not directly perform a calculation.

I will go into more detail on these outcomes.

```
// https://godbolt.org/z/M6en1GMrE
#include <iostream>           // cout
#include <string>
void print(int n) {           // function print for type int
    std::cout << "Number:" << n << "\n";
}
void print(std::string s) {    // same name, different type
    std::cout << "String:" << s << "\n";
}
int main() {
    int number = 10;
    std::string name = "Bilbo";
    print(number);             // calls print(int), number is int
    print(name);               // calls print(string), name is string
```

```

    print(11 + 22);           // expression is int
    print(name + " Baggins"); // expression is string
}

```

Listing 9.1 Types of expressions are important because they determine what happens next.

For example, the result of `11 + 22` is of type `int`—and it is this property that determines that `print(int)` is called and not `print(string)`. It's different with `name + "Baggins"`, whose result is of type `string`—so `print(string)` is called.

You have surely noticed that in both examples `+` was used: once to add numbers and the other time to concatenate strings. The `+` operator is *overloaded* as our own `print`. Applied to `int`, it returns an `int`; applied to `string`, it returns a `string`. How to use operators like `+` was discussed in detail in [Chapter 4, Section 4.3](#).

Either Calculation or Side Effect

It is important to always keep these two aspects of an expression in mind. It is good if an expression *either* performs a calculation *or* has the side effect as its main purpose—and if this is clear from the code or the documentation.

9.2 Types of Expressions

For better clarity, I list here the most important kinds of expressions—not as strict definitions, but in a highly summarized list as an overview. This is good for learning, but not for writing a C++ compiler. I summarize many special cases here that formally do not belong together.

- **Literal**

A literal is a value written directly into the source code—for example, a number like `42` or a text like `"Goofus"`.

- **Identifiers**

There are identifiers for variables, functions, types, and so on, possibly accessed through a *scope* with `::`, such as `std::cout`.

- **Parenthesized expression**

For example, `( 3 + 4 + 5 )`.

- **Function call**

A function call is recognizable by the parentheses (...) that follow it—for example, `sin(x)`.

- **Index access**

An index access uses square brackets [...] instead of the parentheses of a function call, as in `data[3]`.

■ Operators

There are binary operators like + and / in `a+b/2`, or dereferencing with `->` in `it->second`, as well as unary ones like the `++` postfix in `idx++`, but also the `-` prefix in `-10`.

■ Assignment

An assignment is an expression with the equals sign `=` as an operator, such as `x=4`. The value of the right operand is assigned to the left operand.

■ Type casting

Type casting comes in various forms. The C cast uses the target type in parentheses before the value to be cast—for example, `(int)30.1`. Alternatively, you can also use the function notation for this, like `int(30.1)`. A C++ cast also looks similar to a function call, with the target type written in angle brackets—for example, in the expression `static_cast<int>(30.1)`.

If you can master and apply these kinds of expressions, then you are already very well equipped. While the listed expressions appear in C programs, the following are typical of C++ and are therefore powerful tools:

■ Memory management

Allocating with `new` and deallocating with `delete`

■ Lambda expression

An anonymous function that is (usually) defined at the place where it is used

■ Template usage

When the name of a called function can also include angle brackets `<...>`, like `numeric_limits<int>::is_signed`

■ Throw exception

A `throw` followed by an expression, as you have briefly seen in [Chapter 8](#)

9.3 Literals

Literals are fixed, concrete values directly written in the source code. In C++, these can be integers, floating-point numbers, characters, or strings. The basics are the easily recognizable forms `123` and `3.141592` for numbers, `'a'` as a single character, and `"Gügelhupf"` as a string.

It only becomes a bit more complicated because there are prefix and suffix modifiers for these base literals. As you have already seen in [Chapter 4, Section 4.4](#), there are the following forms for integers, floating-point numbers, Boolean values, and strings:

<code>999</code>	<code>0xffff</code>	<code>0777</code>	<code>0b10101</code>	<code>1L</code>	<code>0u</code>	// integer forms
<code>72.75</code>	<code>1e+10</code>	<code>3.141592f</code>				// floating point
<code>'a'</code>	<code>"Text"</code>					// strings
<code>true</code>	<code>false</code>					// truth values, the two bool literals
<code>nullptr</code>						// the only <code>nullptr_t</code> literal

9.4 Identifiers

Many things have a *name*: variables, constants, functions, classes, macros, and many more. These names consist of letters, digits, and the underscore character `_`. By letters, we mean A to Z and a to z; you should not use umlauts and other international characters, although some compilers support them.

An identifier must not start with a digit—so you cannot use `9eye`, for example.

Regarding the underscore, you should not start anything with two underscores `__` and should avoid an underscore followed by an uppercase letter at the beginning, as these forms are reserved for the standard library internally: `__number` and `_Text` can cause problems, for example.

Here are a few valid identifiers—all of which are different, as C++ is case-sensitive:

<code>number</code>	<code>number_with_0</code>
<code>Number</code>	<code>NUMBER_WITH_0</code>
<code>NUMBER</code>	<code>NumberWith0</code>
<code>_number</code>	<code>OneHundredEleven</code>
<code>number_</code>	<code>onehundred11</code>
<code>_number_</code>	<code>one100eleven</code>

Compound Identifiers

When you use functions and similar constructs, C++ combines the complete identifier from multiple elements. Thus, the complete C++ identifier can contain the scope operator `::`, such as `std::cout`. You have also seen that `+` as an operator for C++ is actually called `operator+`, and that is its identifier. The angle brackets for function and class templates are also always written together later, resulting in the entire identifier, such as `vector<int>` or `numeric_limits<int>`.

You don't need to memorize the exact language definition here. What is important is that you get a feel for what constitutes an identifier through practice.

9.5 Parentheses

With parentheses, you can influence the order of calculations within an expression, just like in mathematics. This achieves roughly the same effect as if you were to split the expression into multiple statements. Therefore,

```
int interest = 3 * (4 + 7) * 8;
```

is the same as

```
int intermediateResult = 4 + 7;
int interest = 3 * intermediateResult * 8;.
```

The order of other evaluations (here, `*`) is not affected.

9.6 Function Call and Index Access

A function call in C++ is made with parentheses. Within these, the arguments (or parameters) for the call are listed, separated by commas:

Function (Parameter , Parameter, ...)

Each parameter is again an expression, with function calls without parameters also occurring, and the list is then empty with `()`. And if you look closely, *Function* is also an expression, which usually simply consists of the name of the function to be called:

```
sin(3.141592);
print(6*6+number, "text", name);
justDo();
numeric_limits<int>::max();
[](auto x, auto y) { cout << x+y; } (3, 4); // lambda expression instead of
function
```

It looks similar with *index access*, but you use square brackets there. You cannot list multiple parameters (this will change soon though). However, it is common to apply index access multiple times in a row:

```
data[10]; // index access with number
image[x][y]; // multiple times in a row
addresses["John Doe"]; // access to associative container
```

9.7 Assignment

The assignment is also “just” an expression, in which `=` is the operator:

Expression left side = Expression right side

Here, however, the change to the left operand can be referred to as a side effect. On the right and left, there are again expressions, where on the left side there must be something to which you can assign something—for example, a variable:

```
int number;
number = 26 * 12 + 3;
```

It would make no sense to use a literal on the left side of the assignment, such as `26 = number * 3`. Constants are also out of the question. However, there are cases where a real expression stands there—one that actually computes something.

You have seen that a function can also return a *reference*. You got a more detailed insight into this in [Chapter 7](#). The result of the expression on the left must be a writable *reference*. If the return type of a function is something with `&`, then this function can also be on the left side.

For example, `front()` of `vector<int>` is such a function. It returns a reference to the first element in the vector. It looks like this in principle (though within the `vector` template class and thus simplified here):

```
int& front();
```

Therefore, you can write the following:

```
// https://godbolt.org/z/EY5K6Wsr3
std::vector<int> data(10);           // 10 times 0 in a vector
data.front() = 666;                 // writes 666 to the front position
```

Outlook: Pointers, Arrays, and Iterators

When you learn about pointers, C-arrays, and iterators in [Chapter 20](#), you will write something like this:

```
int numbers[10] = {0};              // 10 contiguous int values initialized to 0
*(numbers+3) = 666;                 // writes 666 to the fourth position in numbers
std::vector<int> data{10};          // also 10 values initialized to 0
*(data.begin()+3) = 777;            // writes 777 to the fourth position
```

The first two lines operate on a *C-array* of 10 `int` values. With `*(numbers+3)`, `numbers` is used as a *pointer*: `+` simply moves three positions forward in the array, and `*(...)` turns the position into a memory location, which is then written to with the `=`. Except that the last two lines involve a class (or to be precise, a template class), it works the same way. Here, it's not called a pointer, but `begin()` returns an *iterator*. You can add three to this and use `*(...)` to access the actual variable.

And because the assignment is an expression, you can embed it almost anywhere. The result of an assignment is the assigned value. You will often see an `if` condition where an assignment is also made (in this example, with an initialization separated by a `;`, as has been possible since C++17):

```
// https://godbolt.org/z/K9c913had
if(int result; (result = read(buffer, 100)) != 0) {
    std::cerr << "Error number " << result << " occurred.\n";
}
```

This is quite a tricky expression:

- `(result = read(buffer, 100)) != 0` calls the function `read()`.
- Its result is assigned to `result =`.
- The result of this expression is then compared to 0. With `!=0`, it checks if it is not equal to 0.
- And this is true or false, so the `if` can decide which way to go.

However, use such assignments within other expressions sparingly. Under no circumstances should you perform “magic” and change values in the middle of a calculation:

```
3 + ((number = 2) * 4) - 6 * (value = (6/2)) // ⚡ not good style at all
```

No one will be able to understand that anymore. It is downright wrong also to use the target of the assignment elsewhere in the expression:

```
3 + (number = 2) * (number = 4) + 6 // ⚡ modified and used
```

This is faulty code of the sort that allows the program to do whatever it wants—not what you want (*undefined behavior*).

Using an Assignment as Part of a Larger Expression Is Dangerous!

Do not reuse the target of an assignment in the same statement. The result is undefined; the compiler does not have to warn you. This applies to all operations with side effects that modify a variable.

9.8 Type Casting

It is easy to explain type conversion for the previously introduced `int`, `bool`, `double`, and `string` and their relatives because there is not much to convert with these types yet. Later, however, this will become an extensive topic that I will return to multiple times. In the context of expressions, I will exemplify the conversion between `int` and `bool` to illustrate the concept.

You can convert an `int` to a `bool` by simply using the type name as a function:

```
int value = 10;
bool yesNo = bool(value); // function notation for type conversion
```

Coming from C, there is another notation for this. It is still very popular, and you will see it often:

```
bool yesNo = (bool)value; // C notation for type conversion
```

There is yet a third variant in C++ to perform the preceding type conversion:

```
bool yesNo = static_cast<bool>(value); // detailed C++ type conversion
```

This is because the other two variants can perform different types of type conversions depending on the type. And in such cases, specifying exactly which conversion—or *cast*—to perform is sometimes indispensable.

In addition to the `static_cast` shown here, there are also `reinterpret_cast`, `const_cast`, and `dynamic_cast`, which I will not cover further here as they are an advanced topic. These are all the built-in cast variants. The standard library adds some things with “cast” in their name, like `any_cast` and `duration_cast`, but I call these *explicit type conversions* to distinguish them from the built-in casts.

Chapter 10

Error Handling

Chapter Telegram

- **Error**

An error is something abnormal or unexpected. An abnormal situation could be, for example, an attempt to open a nonexistent file, and the program will *handle* it; that is, it deals with an exceptional situation. But if something completely unexpected occurs, it could be a program error.

- **Exceptional situation**

Within the scope of this chapter, an exceptional situation is a more or less anticipated but unusual event to which the program responds. This can happen via an exception or through error codes.

- **Program error**

Reaching an undefined state of the program.

- **Programming error (software bug)**

An error in the program that leads to a poor or incorrect result.

- **Exception**

An exception is triggered using `throw`. The current blocks upward of the call chain are exited until a matching `catch` is reached, or the program is terminated if no `catch` matches.

- **Error code**

Alternative to handling errors using exceptions through return value or output parameters (or in extreme cases, a global variable).

- **`throw`**

The act of triggering (throwing) an exception

- **`catch`**

Start of a handling block for exceptions.

Often a program must respond to events that are not what you want them to be: a file is not found, a zero is entered for a calculation period, or so on. There are many ways to deal with such events:

- You can ignore them and expect users to live with the resulting errors. Perhaps you are one of the users yourself.

- You can proactively check the conditions and intermediate results of actions and respond to such behavior in the program: output a message, try again, or terminate the program in an orderly manner.

The latter can usually be done as elaborately as you like. Good handling of unexpected states and errors is a very difficult task.

In C++, you have several options to master this, of which I will discuss two here:

- **Error codes**

Use return values or output parameters as side information about a process. Check these regularly and handle the event on the spot.

- **Exceptions**

Instead of passing the error through a side channel, raise an *exception* at the point of occurrence and *handle* the error elsewhere, probably at a central location for such handling.

First, it must be clear that there are different types of “errors” that need to be handled differently:

- **Undesirable state**

An undesirable state is an error that you should respond to on the spot because you suspect that something might go wrong. Maybe the document loaded by a user is corrupted. You should be as prepared as possible for such things. In the strictest sense, these are not errors because one definition is that an error is *unexpected* behavior. Error codes play a significant role here. But you can also use exceptions for this.

- **Unexpected state**

An unexpected state is an error that you do not want to anticipate everywhere—for example, because a check is costly and the consequences of its occurrence do not justify the increased effort of error handling. Memory is running out; the configuration file that was just read has disappeared. For robust software, you need to respond to more of these unexpected states. I wouldn't say that you *expect* them, but you are simply well prepared. This is where exceptions usually come into play.

- **Program error**

It is quite possible that your program may enter an undefined state in certain situations. This usually leads to a crash, but not always. You can be aware of this when writing the program and intentionally choose to forgo error handling. You might restrict the input in the documentation, but the documentation is part of the program. There you can write “Do not enter more than 100 numbers”, but nothing prevents users from doing so anyway. Not every program error is a programming error, because it is possible that the program was used incorrectly.

- **Programming errors**

Programming errors are mistakes that you have unknowingly introduced and that

produce faulty behavior. Divide by zero? Array one element too small? Your program then has a *bug*. If you are lucky, you can protect yourself from fatal consequences—with error codes, but more often with exceptions.

■ Quality errors

Quality errors are those issues that, while not preventing the correct end result for end users, still impose compromises for users or programmers. For users, this might mean that the program runs too slowly or has difficult-to-use dialogs. For programmers, it could mean that the software becomes hard to maintain, potentially leading to future programming errors. This broad category also involves subjective factors like code readability, testing, documentation, and more.

Quality errors often stem from a lack of adherence to coding standards and best practices, which can result in a disjointed and inconsistent codebase. This inconsistency can make collaborative development more difficult and increase the likelihood of introducing new bugs when making changes. As a result, addressing quality errors is crucial not only for ensuring a smooth and enjoyable user experience but also for maintaining a sustainable and efficient development process.

You can see here that the term *error* encompasses a lot of completely different things. In this chapter, we do not deal with quality errors at all, little with programming and program errors, partly with unexpected states, and mostly with undesirable states. When you handle errors, you are in a way anticipating them—and this chapter is about error handling.

10.1 Error Handling with Error Codes

When working with error codes, use return values and error parameters to convey information about error states. It looks something like the following listing.

```
// https://godbolt.org/z/EWqqGjnnz
#include <iostream> // cout, cerr
#include <fstream>
#include <vector>
#include <string>
using std::vector; using std::string; using std::cout; using std::cerr;

int countWords(const string& filename) { // return negative on error
    std::ifstream file{filename};
    if(!file) { // was there an error opening the file?
        cerr << "Error opening " << filename << "\n";
        return -1; // report an error to the caller using a special value
    }
    int count = 0;
```

```
string word;
while(!file.eof()) {    // not at the end yet?
    file >> word;
    ++count;
}
return count-1;        // one more word was read at EOF
}

bool process(const vector<string>& args) { // return true if all okay
    if(args.size() == 0) { // expecting parameters
        cerr << "Command line argument missing\n";
        return false;    // report an error via return
    } else {
        bool result = true; // for the final result
        for(const string filename : args) {
            cout << filename << ": ";
            int count = countWords(filename);
            if(count < 0) { // special return indicates error
                cout << "Error!\n";
                result = false;    // at least one error
            } else {
                cout << count << "\n"; // output normal result
            }
        }
        return result;    // return overall result
    }
}

int main(int argc, const char* argv[]) {
    bool result = process(    // return value contains error indicator
        {argv+1, argv+argc} );    // const char*[] to vector<string>
    if(result) {                // evaluate return value
        return 0;
    } else {
        cerr << "An error occurred.\n";
        return 1;    // indicate error externally
    }
}
```

Listing 10.1 Respond to many states with different types of return values.

You will often see that states that are not optimal are checked with `if`:

- With `if(!file)`, I check if the file opening was successful.
- With `if(args.size()==0)`, I check if the user has provided at least one command line parameter.

- With `if(count < 0)`, I use the special return value of the function to react to the error.
- With `if(result)` in `main()`, I check the forwarded result.

Each case is handled differently:

- `countWords` returns a count (always zero or greater), so in `countWords` the `return -1;` with the special value of `-1` indicates an error to the caller.
- The return value is checked in `process` at `if(count < 0)`, possibly specially handled, and finally passed outward at `return result;`.
- The test in `process` at `if(args.size() == 0)` can directly lead to a negative return via `return false;`.
- And if an error occurs somewhere, it ultimately reaches `bool result = process();` and is even passed to the callers of the program at `return 1;` from `main()`.

If you react to errors in this way, you pass the error state through function calls using `bool` or `int` codes or—in extreme cases—separate error objects.

I want to quickly explain the argument `{argv+1, argv+argc}` of `process`: The function `process` receives a `const vector<string>&` as a parameter. However, the given argument is an initializer list consisting of `const char*`. The compiler can convert this without problems using the appropriate constructors of `vector` and `string`.

To show you how to use the program, here is how it reacts in different cases:

```
$ touch empty_file
$ echo "Two words" > two_words.txt
$ ./28-codes.x
Command line argument missing
An error occurred.
$ ./28-codes.x empty_file two_words.txt 28-codes.cpp
empty_file: 0
two_words.txt: 2
28-codes.cpp: 252
$ ./28-codes.x two_words.txt DOES_NOT_EXIST.txt empty_file
two_words.txt: 2
DOES_NOT_EXIST.txt: Error opening DOES_NOT_EXIST.txt
empty_file: 0
Error!
An error occurred.
```

First, prepare two files with zero and two words of content using `touch` and `echo`. When you run `./28-codes.x` without arguments, you get an error. Running it with three arguments (`empty_file ...`) works and counts the words in each of the files. A file name that does not exist leads to an error when running with `DOES_NOT_EXIST.txt`.

No matter what you implement—error codes, exceptions, or neither—the behavior of a function in the event of an error is part of the interface. As part of the interface, you must document it just like returns, parameters, and regular behavior. In the standard library, you will see many examples of how to accomplish this documentation.

10.2 What Is an Exception?

You cannot avoid the really important checks for error states: you have to teach the computer what an error is and what is not. But the passing on of codes is tedious and long-winded. You can also centralize handling for errors that occur in multiple places with a different approach.

Instead of generating and passing on error codes, you can *trigger an exception*—or, in other words, *throw an exception*.

```
// https://godbolt.org/z/j8abxvvvo
#include <iostream>           // cout, cerr
#include <vector>
#include <string>
#include <fstream>           // ifstream
#include <stdexcept>         // invalid_argument
using std::vector; using std::string; using std::cout; using std::ifstream;
size_t countWords(const string& filename) { // 0 or greater
    std::ifstream file{};           // create unopened
    // register for exceptions:
    file.exceptions(ifstream::failbit | ifstream::badbit);
    file.open(filename);           // could throw an exception
    size_t count = 0;
    string word;
    file.exceptions(ifstream::badbit); // EOF no longer an exception
    while(!file.eof()) {           // not at the end yet?
        file >> word; ++count;
    }
    return count-1;                // one more word was read at EOF
}

void process(const vector<string>& args) {
    if(args.size() == 0) {         // process expects parameters
        throw std::invalid_argument{"Command line argument missing"}; // trigger
    } else {
        for(const string filename : args) {
            cout << filename << ": " << countWords(filename) << std::endl;
        }
    }
}
```

```

int main(int argc, const char* argv[]) {
    try {                                     // block with error handling
        process(
            vector<string>{argv+1, argv+argc} ); // const char*[] to vector<string>
        return 0;
    } catch(std::exception &exc) {           // error handling
        std::cerr << "An error occurred: " << exc.what() << "\n";
        return 1;
    }
}

```

Listing 10.2 Exceptions are triggered with “throw” and handled with “try-catch”.

At `file.open(filename)`, I no longer check if opening the file was successful, because now something else happens: `file.open()` *triggers an exception* if there was an error. This means that the program is immediately interrupted and the current function—here, `ifstream::open`—is exited. However, this does not happen in the normal way via `return`, but rather on the fly. Also, `countWords()` and `process()` are “unwound” in this manner—until the try-catch block in `main()` is reached, which contains the call to `process()`. There it is stated what needs to be done with `catch(std::exception)` in this case. Here I output an error, and, as in the previous example, `main()` returns a 1.

That was exception handling in a nutshell. For you to really understand it, I need to go into a bit more detail.

10.2.1 Throwing and Handling Exceptions

The keyword `throw` triggers an exception. A value is associated with it, representing the exception. At the beginning of `process()`, you see an example of this: after the `throw`, I create an instance of the `std::invalid_argument` class, which now represents the error.

In the case of `file.open()`, it is not me who triggers the exception with `throw`; `open()` itself does. Therein, in the case of an error, `throw` raises an `ios_base::failure{...}` exception, and `open()` exits this way.

I called this exit above “in passing.” This means that no `return` is used, and it has nothing to do with the declared return type of the function. The return is, so to speak, *replaced* by the value of the exception.

The program does not leave the functions uncontrollably. On the contrary: all objects are carefully cleaned up, as is the usual case when leaving a block normally. So it is still ensured that every created object is completely cleaned up. This includes all types of the standard library and also types that you have created yourself (see [Chapter 16](#)).

Raw Pointers Are Not Properly Unwound

Wait a moment—I need to clarify something: pointers that you have created with `new` are only safe if you have wrapped them in a smart pointer (like `unique_ptr` or `shared_ptr`; see [Chapter 20](#)).

10.2.2 Unwinding the Call Stack

Neither `countWords()` nor `process()` handle exceptions that are thrown within their bounds. That is done by `main()`, which contains a `try-catch` block there. Everything that happens within the curly braces `{...}` of the `try` is now checked when an exception is thrown—and *within* here also means that it could have happened somewhere deep in the basement of a function call.

The check consists of going through the `catch` commands associated with the `try` one by one. The compiler tries to convert the value of the exception into the type specified in the `catch`. In my `catch` of `main()`, it says `std::exception`, which feels responsible for quite a lot—at least for both `std::invalid_argument` and `ios_base::failure`.

My own exception handling now consists of outputting an error message. I add the text with what is attached to the exception value. For example, I get the text with `exc.what()` that I specified as an error message with `std::invalid_argument{...}`.

So that you can also see this program in action, here is the screen output:

```
$ ./28-exc01.x
```

```
An error occurred: Command line argument missing
```

```
$ ./28-exc01.x empty_file two_words.txt 28-exc01.cppp
```

```
empty_file: 0
```

```
two_words.txt: 2
```

```
28-exc01.cppp: 171
```

```
$ ./28-exc01.x two_words.txt DOES_NOT_EXIST.txt empty_file
```

```
two_words.txt: 2
```

```
An error occurred: basic_ios::clear
```

Here too, calling `./28-exc01.x` without parameters leads to an error. If all files can be opened, then the program counts the words. If a file does not exist, then you will get an error again when calling with `DOES_NOT_EXIST.txt`.

10.3 Minor Error Handling

Unlike in the first example, the processing now completely stops; for `empty_file`, you see no result. This is because the exception from `open()` is only caught in `main()`—and thus the `for` loop in `process` was also prematurely terminated.

If you want the same behavior—that is, continuing the processing with the next file—then you must not leave the `for` loop in passing. You need error handling within the loop.

```
void process(const vector<string>& args) {
    if(args.size() == 0) {                // expect parameters
        throw std::invalid_argument{"Command line argument missing"};
    } else {
        for(const string filename : args) {
            cout << filename << ": ";
            try {
                cout << countWords(filename) << "\n";
            } catch(std::exception &exc) {
                cout << "Error: " << exc.what() << "\n";
            }
        }
    }
}
```

Listing 10.3 A “catch” can also be within a loop.

Now the error is caught and handled, and the `for` loop can continue normally with the next file.

The rest of the program remains the same. The `throw std::invalid_argument` is outside the `try-catch` block and reaches `main()` unchanged to be handled.

10.4 Throwing the Exception Again: “rethrow”

If you want *both* options—local error handling, but also error handling in `main()`—then you can use a `throw` without any argument within the `catch` block to rethrow the currently handled exception unchanged (hence the name *rethrow*).

```
try {
    cout << countWords(filename) << "\n";
} catch(std::exception &exc) {
```

```
    cout << "Error: " << exc.what() << "\n";  
    throw; // rethrow  
}
```

Listing 10.4 Using “throw” without parameters rethrows the currently handled exception.

You might need this if you have some specific local cleanup to do but still want to use the main error-handling routine—perhaps to present a sophisticated error dialog to users.

10.5 The Order in “catch”

With the single catch in `main()` from [Listing 10.2](#), I caught all possible exceptions in my program. Both `std::invalid_argument` and `ios_base::failure` can be converted to the specified `std::exception`. However, you can also use two separate handlers:

```
int main() {  
    try {  
        //... Code as before ...  
        return 0; // if everything is okay, return ok as well  
    } catch(std::invalid_argument &exc) { // first handler  
        cerr << "Invalid argument: " << exc.what() << "\n";  
    } catch(std::ios_base::failure &exc) { // second handler  
        cerr << "File error: " << exc.what() << "\n";  
    } catch(std::exception &exc) { // third, very general handler  
        cerr << "An error occurred: " << exc.what() << "\n";  
    } catch( ... ) { // fourth and last handler for the rest  
        cerr << "A strange error occurred\n";  
    }  
    return 1; // you only get here after a catch: report error  
}
```

If an `invalid_argument` exception occurs between `try` and the `catch` blocks, then the `catch` handlers are tried in order—and the first one, `invalid_argument`, matches. The handler is executed and then the entire handler block is exited; the next line is therefore `return 1`. Unlike with `switch`, there is no fall-through to the next handler here.

If the code throws an `ios_base::failure` exception, then `invalid_argument` does not match, and the second handler is tried. It matches and is executed. Even after this handler, the program continues after the last `catch`.

The handler for `std::exception` would have also matched the other two exceptions, but it is not reached in either case. How this conversion works is explained in [Chapter 15](#). You will only reach this point if you extend the program and other exceptions are thrown that match—or if you have overlooked something.

And indeed, we have. An exception that you should always expect is `std::bad_alloc`. This is thrown when `new` requests more memory than is available (see [Chapter 20](#)). This can happen either if your program has really consumed the memory or, often, if an error elsewhere leads to a memory request of size -1 or similar. Usually, `bad_alloc` indicates a serious problem. In any case, `bad_alloc` can also be converted into exception and—theoretically—be thrown by many parts of the standard library. With the `std::exception` handler shown here, you at least catch the error.

The last clause with `catch(...)` captures everything that still comes in here. With exception, I have actually handled everything that can be triggered by and with the standard library, but theoretically, there could be more.

The `catch(...)` clause must be the last one in a `catch` list. Another one could not be reached afterward anyway. Because you cannot give the caught exception a name as in the more specific clauses, the information you can obtain here is very limited. Sometimes, however, there are still important cleanup tasks to be done.

10.5.1 No “finally”

If you come from another programming language (such as Java), then you might now be waiting for the `finally`. Do you miss a block that is executed when leaving a range, whether by exception or regularly? That does not exist in C++; instead, you use the destructor of an object (see [Chapter 16](#)). There, the *resource wrapper* is also described if you really want to simulate a `finally` block.

10.5.2 Standard Library Exceptions

You don't have to deal with an excessive number of exceptions in the C++ standard library. Nevertheless, some can occur almost anywhere. I only want to mention the most important ones here; the documentation of the functions always describes possible exceptions:

- **`bad_alloc`, `bad_array`**
Triggered by `new` and `new[]` when there is a lack of memory.
- **`logic_error`**
These are errors that are more likely to be in the program code and that you might have been able to discover before starting the program.
- **`invalid_argument`**
You already know these exceptions. They belong to `logic_error` and are rarely triggered in the standard library.
- **`out_of_range`**
This exception is also a `logic_error` and is triggered by `vector::at(n)` when `n` is too large.

- **runtime_error**

These include errors that can only occur at runtime.

- **overflow_error, underflow_error**

These are exceptions that you can trigger yourself if your function receives an inappropriate integer argument. The standard library rarely triggers these exceptions. They belong to `runtime_error`.

- **range_error**

You can trigger this exception if your function receives an inappropriate floating-point argument. The standard library rarely throws this exception. It belongs to `runtime_error`.

This list is not exhaustive. You can always see in the documentation of the standard library functions which exceptions they throw.

In larger projects, it is quite common to create your own exception classes. Refer to [Chapter 15](#) for more information. It is best to derive your own class from `std::exception` or `std::runtime_error` for this purpose.

10.6 Types for Exceptions

What else can there be besides `std::exception` and everything that can be converted into it? In principle, you can use values of any type with `throw`. However, it is recommended to use the exception types from the standard library whenever possible.

If that is not possible, then use `int`, `double`, and `string` for `throw`.

```
// https://godbolt.org/z/67cd5bhhE
#include <string>
#include <iostream> // cout
using std::string; using std::to_string; using std::cout;
void triggerError(int errorCase) {
    try {
        if(errorCase < 10) throw (int)errorCase;
        else if(errorCase < 20) throw 1.0/(errorCase-10.0);
        else throw string{"Error " + to_string(errorCase)};
    } catch(int eval) {
        cout << "int-error: " << eval << "\n";
    } catch(double eval) {
        cout << "double-error: " << eval << "\n";
    } catch(string eval) {
        cout << "string-error: " << eval << "\n";
    }
}
```

```
int main() {
    triggerError(3); // Output: int-error: 3
    triggerError(14); // Output: double-error: 0.25
    triggerError(50); // Output: string-error: Error 50
}
```

Listing 10.5 You can also throw exceptions of other types.

There is usually no reason to use types with `throw` that cannot be converted to `std::exception`. For understanding how exceptions work, it is useful to know that `std::exception` is not really special when it comes to exception handling; it just comes with some helper functions, like `what()`.

10.7 When an Exception Falls Out of “main”

If you do not handle a thrown exception at all, so that it reaches `main()` and is not caught there either, then your program will terminate. Most of the time, you will still see a message. Therefore, it is quite common to install only the necessary handlers in `main()`. Those that would do nothing anyway you do not implement at all. The objects you have created will all be properly cleaned up before the program terminates.

However, seeing a message is not mandatory for the system. So if you at least want to see the error type and text, then wrap the entire code in `main` with a simple error output.

```
// https://godbolt.org/z/TsP33njar
#include <iostream>
#include <stdexcept> // exception

int main() {
    try {
        // ... your other code ...
    } catch(std::exception& exc) {
        std::cerr << "main: " << exc.what() << "\n";
    }
}
```

Listing 10.6 At least output the error type and text instead of letting your program tumble out of “main”.

Chapter 11

Good Code, 2nd Dan: Modularization

So long as you write a small program that fits into a single source file, a C++ project is simple. It becomes more difficult when you need to split your program to increase clarity. What goes where?

In this chapter, I use classes for explanation. If you are working through the book sequentially, please forgive the jumping ahead. But this should not distract from the principles of modularization. You can easily return to the dan chapters later.

C++20: Modules

Since C++20, *modules* enable a clean separation of declaration and implementation. They also allow the compiler to import faster than with `#include` included program parts. Look out for the new `module` and `import` keywords.

Although a lot has changed regarding modularization, in practice, much will remain the same: `#include`-based modularization will stay with us for various reasons for a long time. What you learn in this chapter will also be useful when working with modules.

11.1 Program, Library, Object File

When the compiler (or linker) assembles the executable program, it usually takes your translated source code and combines it with other libraries available in the system.

Each `*.obj` file represents a `*.cpp` file translated by the compiler that you have written. The `*.dll` files are libraries present on your system that your program is aware of but does not need to fully integrate (hence the dashed line in [Figure 11.1](#)). The [Figure 11.1](#) `*.lib` files are also libraries, but ones that your program integrates fully—or at least parts of them. For example, libraries of the C and C++ compilers as well as a game library are shown here. You may have acquired the latter from a third-party vendor so that you don't have to develop essential parts of the program yourself.

As a Unix user, you are more familiar with the extensions `*.o`, `*.so`, and `*.a` instead of `*.obj`, `*.dll`, and `*.lib`, but the same applies to all systems: the final program is assembled from multiple libraries.

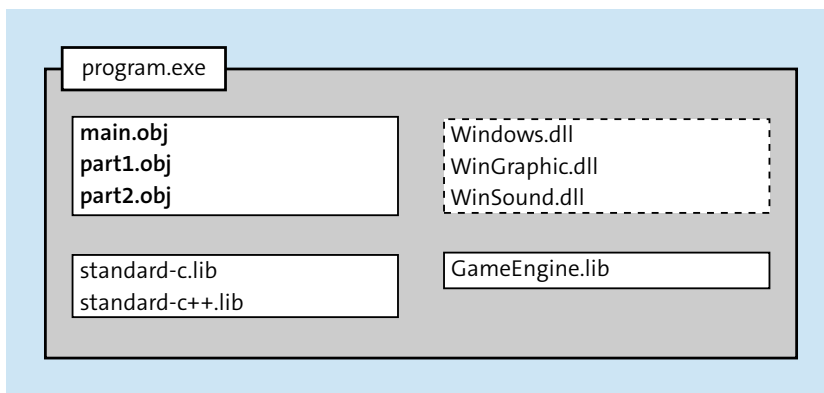


Figure 11.1 Apart from your code, a program consists of several libraries.

Each library usually consists of individual modules. For example, the C++ standard library, from your perspective as a user, consists of several headers. Sometimes you include `#include <vector>`, sometimes `#include <iostream>`. These headers reflect that the library was probably not developed as a single huge `*.cpp` file. The modules were combined into a single library for your convenience, so you only need to specify this one `*.lib` file to the compiler. The compiler then picks out the things that are actually used by itself.

11.2 Modules

Your task now is to sensibly divide the modules of your part of the program for your own project. For this, you need to answer the question, for example, “What belongs in `main.obj`, `teil1.obj`, and `teil2.obj`?” And along with that: “What belongs in the `main.cpp`, `teil1.cpp`, and `teil2.cpp` source code files?”

`main.cpp` should probably contain `int main()`. Instead of naming the file `main.cpp`, `programm.cpp` is also a reasonable name if your program will later be called `programm.exe`.

Otherwise, you should break your project into parts based on functionalities. Each part goes into a separate set of `*.cpp` files. Each `*.cpp` file, in turn, is complemented by a similarly named `*.hpp` file. The exception is `main.cpp`, for which you normally do not need to create a header file.

Interface: Each `.cpp` File Also Gets a `.hpp` File

Write a similarly named `*.hpp` header file for each `*.cpp`. Declare in it all the functions and other things that are directly used by other `.cpp` files: these form the *interface*. Intentionally keep certain elements only in the `*.cpp` file—specifically, those needed solely to implement the interface.

It is a good standard practice to have a corresponding `*.cpp` file for every `*.hpp` file. But there are more exceptions to this rule. Sometimes you combine headers for simplicity; sometimes the interface is also the implementation or contains only constants. You might consider separating such headers from the normal ones, naming them differently, or managing them in a different directory.

11.3 Separating Functionalities

That's easier said than done, because often the boundaries of functionalities are fluid.

When you break down your program into `*.cpp` and `*.hpp` files, the consequence is that the various `*.cpp` and `*.hpp` files include other `*.hpp` files via `#include`.

It gets bad when every file needs all the other files. Try to keep the number of required `#include` directives small.

Cyclic dependencies are really tricky—where `A.hpp` includes `B.hpp`, but in `B.hpp` you then realize that you actually need `A.hpp` first. You solve this by separating one (or both) headers so that only the `*.cpp` file needs the other header.

I therefore recommend following these rules:

- Separate your program into layers. One layer may only use things from the layers below. But skip as few layers as possible:
 - An `*.hpp` file should ideally only include something from the direct layer below. Another part of the program that wants to include this header will then also need all the others, and that can quickly become many. But the fewer there are, the better.
 - A `*.cpp` file can be more permissive. The disorder is localized to compiling this source file, so it doesn't have such a wide impact.
 - There are exceptions. In [Figure 11.2](#), `const.hpp` has the role of consolidating central declarations. Probably every header needs these, and then it's good to have them in one place, in a single file, and not scattered across multiple files.
- Avoid cyclic dependencies:
 - A `*.cpp` file must always include “its” `*.hpp` file anyway.
A `*.cpp` file may use any other `*.hpp` file at this level. `teil2.cpp` may include `teil1.hpp`, as shown in [Figure 11.2](#). There would also be nothing against `teil1.cpp` in [Figure 11.2](#) using `teil2.hpp` at the same time.
 - This does not apply to `*.hpp` files. Include other headers of the same level only in exceptional cases. And if you do, strictly only *in one direction*. In the illustration, `teil1.hpp` may thus use `teil2.hpp`, but not vice versa.

If you follow these rules as much as possible, you will definitely avoid cyclic dependencies. The layering model is also beneficial to keep the number of required includes in later modules lower. It is not a panacea, but it is a step in the right direction.

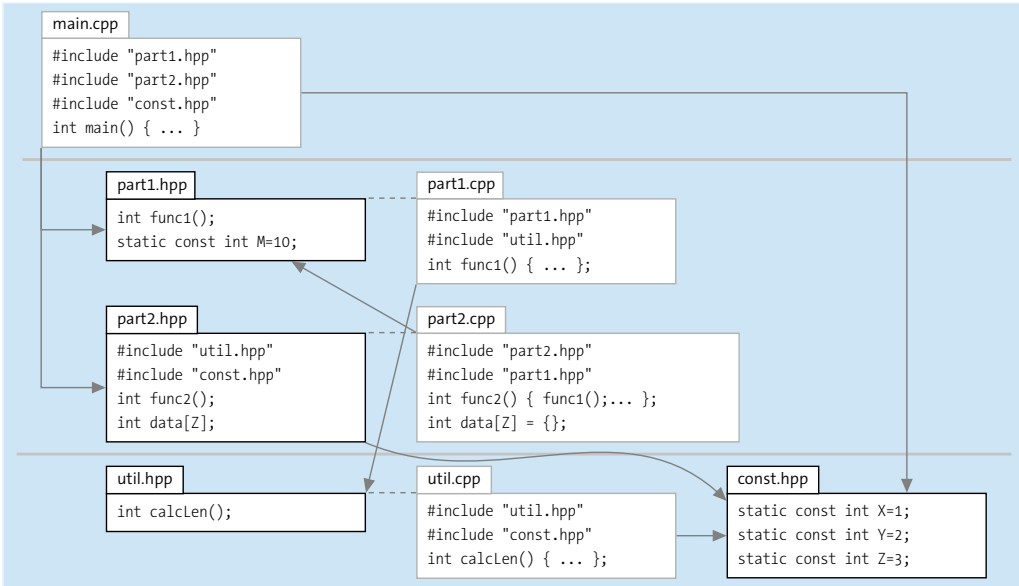


Figure 11.2 Divide the components of your program into layers to avoid cyclic dependencies.

Modularization and OOP

When you work cleanly with classes and traditional object orientation, it is a natural cut to create one “module” per class. In [Chapter 12](#), the description of object-oriented programming (OOP) in C++ begins, and in [Chapter 14](#), you will see a practical example of splitting a project into modules with classes.

11.4 A Modular Example Project

At www.rheinwerk-computing.com/5927, you will find the project in the 15p-unittest-200 directory. It contains the files and directories listed in the following table. For easier experimentation, I put all the code files into one Compiler Explorer link at <https://godbolt.org/z/zj91TshnY>.

File/Directory	Description
README.md	Description of the project
Makefile	Main Makefile—for example, for make all

Table 11.1 Structure of an example project.

File/Directory	Description
all.mk	Definitions for all Makefiles
include/	Include files to be delivered with the library
- qwort/	All public includes of the library
- - qwort.hpp	Main include file of the library
src/	All implementation files
- lib/	Implementation files of the lib
- - Makefile	Makefile for library
- - qwort.cpp	Implementation of the qwort class
- - impl_multimap.hpp	Definition of a helper class
- - impl_multimap.cpp	Implementation of the helper class
- prog/	Executable programs
- - Makefile	Makefile for the programs
- - query.cpp	Simplest example program
- - dictfile.cpp	Example program
- - pg40185.txt	Example text for indexing

Table 11.1 Structure of an example project. (Cont.)

I also use this example project later in [Chapter 14](#) on good code to demonstrate unit tests.

The project consists of a library that I pretend to deliver to a customer. For testing and demonstration purposes, I want to provide the customer with two small example programs to show how to use the library. Whether these example programs are given to the customer as source code or only as a finished program is irrelevant. The customer will receive the header files and the compiled library, but not the source code.

This example library shall mainly consist of a class that implements an index. It will function similarly to a dictionary: sequentially adding multiple entries that then make up the index. With this, you can query the index with a search pattern to check if a specific entry exists. An exact search would be boring, so we search “approximately.” The search should *return the best match*.

I don't want to tell the customer anything more. What the “best match” is should be an implementation detail. Here, I want to use *q-grams* for the search, where I choose the

length of q to be 3. This means that the search finds the entry where most of the three-character substrings of the search pattern match an index entry.

If the index contains, for example, *Graumaus* and *Nagetierstamm*, then with the *Anagramm* query you will find the latter, because both contain *NAG* and *AMM*, while the former does not have any three-letter group in common with *Anagramm*. I don't want to say more about the algorithm as an example here, because it is really a side issue in this chapter. (While translating this chapter, I discovered that German is an excellent language to do q -grams with!)

If you call `make all` in the main directory, then the `qwort.a` library in `src/lib/` and the programs in `src/prog/` will be built one after the other.

The library here consists only of a minimal set of files: a header file that is to be delivered with the library, as well as the implementation files. These, in turn, consist of two `.cpp` files, `qwort.cpp` and `impl_multimap.cpp`, as well as an internal `.hpp` file, `impl_multimap.hpp`, which is not intended for delivery. Therefore, it is also located in `src/lib/`. If you deliver the library to a customer who uses a compiler compatible with yours, for static linking it is sufficient to deliver the `qwort.a` library file and on Windows `qwort.lib` together with the `qwort.hpp` header file. If you want to support dynamic linking too, you provide a `qwort.so` or `qwort.dll` library file.

The Header File Must Match the Library File

It is crucial that you (and your customers) use the exact same header file that was used to build the library file when using your library. If you have different versions in use, the best thing that can happen is a compiler or linker error; in the worst case, the behavior will be unpredictable.

The main `qwort.hpp` header file is shown in [Listing 11.1](#).

```
// https://godbolt.org/z/e5GWbbq4Y
#ifndef QWORT_H // header guard
#define QWORT_H
#include <string>
#include <memory> // unique_ptr
namespace qw { // library namespace
    int version();
    namespace impl_multimap {
        class index_impl;
    }
    class index {
        using index_impl = impl_multimap::index_impl;
    public:
        index();
        ~index() noexcept; // needed for pimpl
    };
};
```

```

    index(const index&) = default;
    index(index&&) noexcept = default;
    index& operator=(const index&) = default;
    index& operator=(index&&) = default;
public:
    void add(const std::string &arg);
    size_t size() const;
    std::string getBestMatch(const std::string& query) const;
public:
    // public for tests
    std::string normalize(std::string arg) const;
private:
    const std::unique_ptr<index_impl> pimpl;
};
} // namespace qw
#endif // header guard

```

Listing 11.1 The main header file of the qwort.hpp library.

The `add` and `getBestMatch` methods of the `index` class are the actual interface of the library:

- First, you create an empty index with the default `index()` constructor.
- Then, you add entries one by one using `add(const string&)`.
- When you are done, you can query as many times as you want with `getBestMatch(const string&)` for the best matching entry that you previously added.

11.4.1 Namespaces

Everything intended for the export of the library is defined in its own namespace `qw`. Therefore, the header also begins with a namespace `qw`, and all declarations and definitions are enclosed within it.

This is followed by a forward declaration of the `impl_multimap::index_impl` class. A namespace is opened for it as well. All internal library components that should be as hidden as possible are contained within it. However, the name of the `index_impl` class, which the `index` class uses further down, is needed here.

Switching Implementations with Namespaces

By the way, the namespace is called `impl_multimap` because it contains only one of several possible implementations. In another directory, you will see the `15p-unittest-100` project in the download sources, which defines an `impl_simple_map` namespace with an `index_impl` class contained within it. This is a different, simpler version of an index implementation.

In principle, you could run both namespaces in parallel and easily switch the implementation used in the main class. However, this is just a suggestion here, which is why I only show it in a rudimentary way. If fully developed, this switchability would involve a bit more programming effort and is not the focus of this chapter.

Now follows the actual main class of the `class index` library. At its core, it contains a `pimpl` field of the `index_impl` class (as a pointer). With this “trick,” implementation details of the `index` interface class can be separated even better. Sometimes this is called the *pointer to implementation* (or *Pimpl*) *pattern*.

To avoid bothering users of the class with implementation details (or to keep them secret), all methods of the class are declared only. They are defined later in the `qwort.cpp` implementation file.

Normally, as you will see in [Chapter 17](#), you would not define a destructor, copy constructor, move constructor, assignment operator, or move operator (the “big five”). The compiler would generate them optimally for you. Unfortunately, that is not possible here: the compiler would need the *definition* of the `index_impl` class for this, because how else would it know what to do with `unique_ptr<index_impl> pimpl`? But so far, we have only *declared* it as a forward declaration. Therefore, the compiler can only help us here if we let it generate the code in the `qwort.cpp` file—especially for the constructor, destructor, and move constructor. You will see how this works shortly. We can simply let the compiler generate the other three here with `= default`. However, we must list them here, because according to C++ rules, they would otherwise be completely missing as we have defined at least one other member of the “big five” family.

11.4.2 Implementation

I chose to separate this chapter into the `index` interface class and the `index_impl` implementation class for two reasons. On the one hand, I wanted to present the Pimpl pattern in practice; on the other hand, this example should not be trivial. With this separation, I have an excellent reason for a second class and a second `*.cpp` file. I can then also handle both later in a second test module.

The header file of the interface class also includes the `*.cpp` file with the implementation, as you can see in [Listing 11.2](#).

```
// https://godbolt.org/z/TMExE8doz
#include "qwort/qwort.hpp" // self
#include <map>
#include <algorithm>        // transform
#include <cctype>           // toupper
#include "impl_multimap.hpp"
using std::map; using std::string;
```

```

namespace qw {
    int version() {
        return 1;
    }
    // administration
    index::index()
        : pimpl{ new index_impl{} }
    { }
    index::~index() noexcept = default;
    // interface
    void index::add(const string &arg) {
        pimpl->add(normalize(arg), arg);
    }
    size_t index::size() const {
        return pimpl->size();
    }
    string index::getBestMatch(const string& query) const {
        return pimpl->getBestMatch(normalize(query));
    }
    string index::normalize(string str) const {
        using namespace std; // begin, end
        transform(begin(str), end(str), begin(str), [](char c) {
            return ::isalpha(c) ? ::toupper(c) : '#';
        });
        return str;
    }
} // namespace qw

```

Listing 11.2 The interface class forwards all calls to the implementation class.

First, all include and using directives are tended to. A sensible order is to start with the *.hpp file that belongs to the current *.cpp file—in this case, "qwort/qwort.hpp". Note that the file name is enclosed with "" instead of <> because the include is part of our own library. In addition, qwort.hpp is located in its own subdirectory qwort/, which is a good idea in larger libraries with multiple header files.

The implementation of the index class now requires the complete definition of the index_impl class; therefore, its header file must also be included with #include "impl_multimap.hpp".

Because everything that is part of this library is in its own namespace, everything in this file is also enclosed in namespace qw.

The version() free function is quickly implemented. Then the implementations of all contents of index follow. First, these are the constructors and destructors that have not yet been defined in *.hpp. At this point, the index_impl class is fully known, so we can

ask the compiler to generate these constructors and destructors with `= default`. In the default constructor, that is not enough: the compiler would initialize `pimpl` with `nullptr`, but we need a new `index_impl`.

The `add()`, `size()`, and `getBestMatch()` methods mainly delegate their calls to `pimpl`. Only `normalize()` is called beforehand each time, because `index_impl` does not handle that, as the implementation of `index_impl` is contained in the current file and therefore included last. Using `transform()`, all characters of `str` are individually converted to uppercase if they are letters; if not, a `#` is inserted. This is done in the fourth parameter of `transform()` using the `isalpha()` and `toupper()` C functions for each individual character. This is a lambda—essentially, a function on the spot.

The `normalize()` function ensures that for e.g., `E#G#` is stored in the index along with e.g., `E`. If you later search for e.g., then another `normalize()` ensures that it can also be found in the index.

The `index_impl` helper class is defined in the corresponding header, which is included at the top of the file. This is, as mentioned, necessary so that the implementation of `index` can take place. However, it is an implementation detail of the library, and thus `impl_multimap.hpp` is not found in the directory of `qwort.hpp`, as it is not shipped with it. It is located right next to `qwort.cpp` and `impl_multimap.hpp`. You can see the source code itself in [Listing 11.3](#).

```
// https://godbolt.org/z/ob9voovMx
#include <string>
#include <string_view>
#include <vector>
#include <map> // multimap
namespace qw::impl_multimap {

using std::vector; using std::multimap;
using std::string; using std::string_view;

class index_impl {
    vector<string> entries;
    multimap<string, size_t> qindex;

public:
    void add(string_view normalized, string_view original);
    string getBestMatch(string_view normalized) const;
    size_t size() const {
        return entries.size();
    }
}
```

```
private:
    vector<string> qgramify(string_view normalized) const;
    static constexpr size_t Q = 3;
    static const std::string PREFIX;
    static const std::string SUFFIX;
public: //test interface
    vector<string> _qgramify(string_view n) const { return qgramify(n); }
    static size_t _q() { return Q; }
    static std::string _prefix() { return PREFIX; }
    static std::string _suffix() { return SUFFIX; }
};
} // namespace qw::impl_multimap
```

Listing 11.3 Header of the implementation class.

Because the focus here is not really on the algorithm, I will keep it brief. In the `entries` vector, the index stores all original (non-normalized) entries so that the search results can also be displayed. Thus, each entry has a number in the vector. In `qindex`, each three-word key has all the entry numbers in the index. For example, if you later search for `HELLO`, then the `multimap` will be examined sequentially for the keys `HEL`, `ELL`, and `LLO`. The values are numbers in the `entries` vector—that is, indexes of the entries there.

`add()` is used to insert new entries into the index, while `getBestMatch()` finds the best matching entry. Both methods expect an already normalized parameter. And both methods use the `qgramify()` helper function to split a string into its `q`-grams.

For testing purposes, I have also made a `public` section here again. Unlike `normalize()`, which I also made `public` specifically for testing, I have left the parts to be tested here `private`. Instead, all things to be tested later are enriched with public access methods. As a convention, these methods start with an underscore `_`. I will go into this in [Chapter 14](#) on testing, but I need to mention it here so that you are not surprised by this `public` section.

I also want to mention that I have placed the entire class not only in the `qw` namespace but also in its own `impl_multimap` namespace. The class itself is called `index_impl`. If you want to provide an alternative, simply put your class in its own namespace and name it `index_impl`.

No magic happens in the header; you are more likely to see it in the implementation, as shown in [Listing 11.4](#).

```
// https://godbolt.org/z/hWan33Go9
#include "impl_multimap.hpp" // header for this file
#include <map>
#include <string>
#include <string_view>
```

```
namespace qw::impl_multimap {

using std::vector; using std::multimap; using std::map; using std::string;
using std::string_view; using namespace std::literals::string_literals;

void index_impl::add(string_view normalized, string_view original) {
    /* TODO: Check for existence in 'entries' */
    const auto pos = entries.size(); // Index of the new entry
    entries.push_back(string(original));
    auto qgrams = qgramify(normalized);
    for(const auto& qgram : qgrams) {
        qindex.insert( make_pair(qgram, pos) );
    }
}

string index_impl::getBestMatch(string_view normalized) const {
    auto qgrams = qgramify(normalized);

    /* hits stores which words were hit how often */
    map<size_t, size_t> hits; /* 'entries-index' to 'hit-count' */
    size_t maxhits = 0; /* always: max(hits.second) */
    for(const auto& qgram : qgrams) {
        auto [beg, end] = qindex.equal_range(qgram);
        for(auto it=beg; it!=end; ++it) {
            hits[it->second] += 1; /* hit-count of the entry */
            if(hits[it->second] > maxhits) { /* simpler max-search */
                maxhits = hits[it->second];
            }
        }
    }

    /* Search first entry with maxhits. Improvement possible with PriorityQueue */
    for(auto const &hit : hits) {
        if(hit.second == maxhits) {
            return entries[hit.first];
        }
    }

    /* only reached if entries is empty */
    return "";
}

const string index_impl::PREFIX = string(Q-1, '^');
const string index_impl::SUFFIX = string(Q-1, '$');
```

```

vector<string> index_impl::qgramify(string_view normalized) const {
    auto word = PREFIX+string(normalized)+SUFFIX; /* trick for better q-grams */
    vector<string> result {};
    auto left = word.cbegin();
    auto right = std::next(word.cbegin(), Q); /* okay: |"^^"|" $"| => 3 */
    for( ; right <= word.end(); ++left, ++right) {
        result.emplace_back(left, right);
    }
    return result;
}
} // namespace qw::impl_multimap

```

Listing 11.4 The header of the implementation class.

As I said before, for this chapter, exactly how the algorithm is implemented is not so interesting. For the module, it is much more important that everything works as it should externally. Nevertheless, I would like to say a few words about the structure of this implementation file.

As always, I include the header of what is implemented in this file as early as possible. Then, only the necessary headers of the standard library follow.

The defined methods all belong to the `qw::impl_multimap` namespace, which is opened at the beginning and closed at the end.

`add()` stores the entry in `entries` and `qindex` after it has been dissected using `qgramify()`. In one aspect, this implementation of `add()` behaves poorly: if you store exactly the same entry twice, you will never find one of them again. So it would be good if `add()` checked whether the entry already exists before storing it, or if a search would find not only the best match but somehow also several good matches. However, I leave solving this problem to your tinkering spirit.

`getBestMatch()` also breaks down the search pattern using `qgramify()` into its components. In `hits`, it counts how often each original word was matched. At the end, the entry with the highest number of matches is selected and returned.

In `qgramify()`, I use a simple trick to slightly increase the number of generated q-grams: the entry or search pattern is framed by `^^` and `$$` before being broken down. As a result, the list of q-grams for words of length three like `NYC` includes not only `NYC`, but also `^N`, `^NY`, `YC$`, and `C$$`, thus having better chances of being matched.

All in all, creating q-grams for a similarity index is neither a very difficult task nor a perfect solution. There are many algorithms that achieve better results with more sophisticated methods. But for its simplicity, the q-gram index delivers surprisingly good results.

This entire functionality is then packaged into the `qwort.a` library. On other systems, it may also be called `qwort.so`, `qwort.lib`, or `qwort.dll`. Together with the `qwort/qwort.hpp` header and documentation, this is the minimum delivery package for the customer.

11.4.3 Using the Library

Because you are my customers today, I will demonstrate how you can use the library in a program before we move on to the tests. [Listing 11.5](#) is pretty much the simplest example program to demonstrate the library.

After the `include` and `using` directives, the `main` program follows immediately. First, two example entries are stored in the index. Then follows a loop over all command line arguments of the program. For each argument, a search for the best match is performed, which is then output.

```
// https://godbolt.org/z/zj91TshnY (includes all files)
#include <cstdlib> // EXIT_SUCCESS
#include <iostream> // cout
#include <vector>
#include <string>
#include "qwort/qwort.hpp"

using std::cout; using std::vector; using std::string;

int main(int argc, const char* argv[]) {
    cout << "qwort version " << qw::version() << "\n";

    /* Build index */
    qw::index myindex{};

    /* - Demo data */
    myindex.add("Germany");
    myindex.add("Greece");

    /* Generate queries */
    vector<string> args(argv+1, argv+argc); // iterator-based initialization
    for(auto &querystring : args) {
        cout << "Searching for '" << querystring << "'... ";
        const auto match = myindex.getBestMatch(querystring);
        cout << match << "\n";
    }
    return EXIT_SUCCESS;
}
```

Listing 11.5 Pretty much the simplest example program of the library.

The compiler packs this `query.cpp` source file with `qwort.hpp` (because it is included) and the `qwort.a` library together into the `query.x` program. The Makefile essentially says the following (somewhat obfuscated by variables):

```
g++ -o query.x query.cpp -lqwort -I../include -L../lib
```

This means something like, “Create `query.x` from `query.cpp`, use the `qwort` library, and look for includes and libraries in the specified paths.”

If you run this program as follows, you will get this output:

```
$ ./query.x armada green
qwort version 1
Searching 'armada'... Germany
Searching green'... Greece
```

In the downloadable content at www.rheinwerk-computing.com/5927, you will find another simple example in `code/15p-unittest-200/src/prog/dictfile.cpp`. There, I create the index from lines of any text file. With additional command line parameters, you can then output the best matching lines.

PART II

Object-Oriented Programming and More

It's time for you to start building your own data types. This is the real strength of C++.

You can simply bundle multiple pieces of data together so that you can treat them as one. More importantly, you can design classes where data and behavior form a single unit. This ultimately leads you to learn about object-oriented inheritance.

In this part, you'll also learn more about constant and static values, as these become particularly interesting in the context of classes.

Chapter 12

From Structure to Class

Chapter Telegram

- **Aggregate**

An aggregate is a simple bundling of multiple data fields into a new type—specifically, a `struct` that only has data fields and methods. A C-array is also an aggregate.

- **`struct`**

A `struct` is a bundling of data fields. It may also have constructors, and private and static elements. It can be a simple aggregate or a more advanced structure.

- **`class`**

A `class` is very similar to a `struct`, except that `class` starts with private visibility; in common usage, a `struct` is more data-focused, while a `class` is more behavior-oriented.

- **Structure**

A structure can be more than an aggregate; for example, it can have constructors. When I speak of a *structure*, I usually refer to both `struct` and `class`.

- **Data field, member variable, or attribute**

A variable within a structure.

- **Method, member function, or instance function**

A function that is bound to its structure.

- **Member**

Generic term for data fields and methods of a class.

- **Constructor**

A special method for initializing a structure.

- **Inline**

Give the compiler a hint to insert the code directly instead of using a function call.

- **Include guard**

Avoid double inclusion.

- **`public` and `private`**

Protect methods and data fields from external access.

- **Encapsulation**

The clean separation of interface and implementation; possible in varying degrees.

- **Class as value**

Use the instance of a class as a value parameter (call-by-value) or as a return value.

- **Copy elision**

Compiler technique to avoid unnecessary copying of a return value.

- **Implicit type conversion**

Within an expression of type A, the compiler requires another type B and generates the desired B using its single-argument constructor B(A).

- **Explicit type conversion**

When you explicitly call the single-argument constructor in an expression.

- **Conversion method**

A method of the form `operator TargetType()`, which allows you to convert an object to `TargetType`.

- **const method**

A method that does not change the state of the instance; it is marked with `const` at the end.

- **Type alias**

Using `using newType = oldType` or `typedef oldType newType` to introduce an alternative notation for a type—also locally within a class.

- **Type inference and auto**

With `auto`, you can let the compiler deduce the type of a variable during initialization.

A custom data type can, for example, hold data about a `Person`. You then don't always have to gather the individual elements and pass them to functions one by one; you can bundle several into an *aggregate*. Use `struct` for that.

```
// https://godbolt.org/z/8YqoTEs58
#include <string>
#include <iostream>                // cout
#include <format>
using std::string; using std::cout; using std::format;
struct Person {                   // defines the new type Person
    string name_;
    int age_;
    string city_;
};                                // closing semicolon
void print(Person p) {            // entire Person as one parameter
    cout << format("{} ({} from {})\n",
        p.name_, p.age_, p.city_); // access via dot
}
int main() {
    Person john {"John", 45, "Boston"}; // initialization
    print(john);                       // call as a unit
}
```

Listing 12.1 Creating your own data type with “struct”.

With `struct Person`, the new `Person` type is defined. It consists of the union of the three listed elements. As always, the type is named first, followed by the name. Under this name, you can then access the elements with a dot `.`, as shown in `p.name_`.

I have ended the names of all elements here with an underscore `_`. This is just a convention to distinguish them from global variables and parameters.

As you can see in `void print(Person p)`, the parameter of the `print` function is now the entire person. You do not need to pass the `name_`, `age_`, and `city_` elements separately. `format`, by the way, was newly introduced in C++20.

Terminate Type Definitions with a Semicolon

You terminate the definition of a new type with a semicolon. Unlike *compound statements*, you place a `;` after the closing curly bracket here. Forgetting the semicolon is a common oversight.

In structures, it is called a *definition* when you list the entire content of the structure and a *declaration* when you only mention the name. This can sometimes be necessary when structures use each other.

```
// https://godbolt.org/z/zPbK16zWE
#include <memory>           // shared_ptr
#include <vector>           // vector
struct Employee;           // class declaration
struct Boss;               // class declaration
struct Employee {          // class definition
    std::shared_ptr<Boss> boss_; // pointer to Boss
    void print() const;       // method declaration
};
struct Boss {              // definition
    std::vector<std::shared_ptr<Employee>> employees_; // Pointer to Employees
    void print() const;     // method declaration
};
void Employee::print() const { // method definition
    // ...
}
void Boss::print() const {    // method definition
    // ...
}
```

Listing 12.2 The declaration of a class initially only mentions its name, while the definition contains all its elements.

In this example, there are two classes that use each other. You can only use `Boss` in `Employee` if you have previously *declared* it.

Note that *declarations* method can occur in such a class *definition*. This makes the term *class definition* somewhat confusing.

12.1 Initialization

The initialization can now be done like this, `Person john { "John", 45, "Boston" }`, with the listing of the initialization values of the subelements in curly brackets. Alternatively, you can also use the empty list `{}` for initialization; then all elements are *value-initialized*—that is, with zeros for numbers and with the empty string `""` for strings (see [Chapter 4, Section 4.4.2](#)).

If you do not specify enough elements in the initializer list, the remaining ones will also be value-initialized. However, you should avoid this as it looks like an oversight and always raises eyebrows. If you specify too many elements, the compiler will complain.

But you must not omit the initialization entirely. For example:

```
Person john;
```

This does initialize the string attributes `name_` and `city_` (they are full-fledged classes), but not `age_`, as it is of the built-in data `int`; `age_` could receive a random value.

Note that you can also use an equal sign when listing all elements:

```
Person john = { "John", 45, "Boston" };
```

However, I recommend sticking to the form without the equal sign, which is more common.

Starting from C++20, you can also name the fields you want to set during initialization; this is the *designated initializer*. If the order of the fields in the class changes, the compiler will notify you.

```
// https://godbolt.org/z/K8WePajvM
Person john1 { "John", 45, "Boston" }; // correct
Person john2 { "Boston", 45, "John" }; // oops, swapped, and no one notices
Person jack { .name_ = "Jack", .age_ = 23, .city_ = "Dallas" }; // okay
Person jimmi { .name_ = "Jimmi", .age_ = 48 }; // okay, not all specified
Person carl { "Carl", .age_ = 53 }; // ✗ all designated or none
Person paul { .age_ = 34, .name_ = "Paul", .city_ = "Reno" }; // ✗ swapped
Person jim(.name_="Jim", .age_=34, .city_="NYC"); // ✗ not with parenthesis
```

Listing 12.3 With designated initializers, you specify the elements to be set.

For `paul`, the designated initializers must be specified in the correct order. As with `jimmi`, you can also omit fields, which will then be value-initialized. However, you must not mix fields with and without designated names, as with `carl`.

Warning about Parentheses

You can also write the initialization elements in parentheses, like `string name("x")`. However, if you have no element for the list, then you must not use the parentheses: `string name()` unfortunately means something else (a function declaration). Therefore, it is better to consistently initialize with curly brackets: `string name{}` is a correct initialization.

In a few cases, initialization with curly brackets and parentheses means something different; `std::vector` is one such exception.

12.2 Returning Custom Types

Returning a `Person` is just as simple as using it as a parameter:

```
// https://godbolt.org/z/rvjPorT1o
// Snippet
Person create(string name, int age, string city) { // return type
    Person result {name, age, city};
    return result;
}
int main() {
    Person john = create("John", 45, "Boston"); // store return value
    print(john);
}
```

There is actually nothing new for you here, except that you can see that you can return variables of your own type just as you can with built-in types and those of the standard library.

It may still be worth noting that just like with the types you already know, you don't always need a variable to store the value. You can use your new type as part of an expression. This then creates *temporary variables* of your type as usual:

```
// https://godbolt.org/z/ME76bfbWa
Person create(string name, int age, string city) {
    return Person{name, age, city}; // returned directly
}
int main() {
    print(create("John", 45, "Boston")); // return value used directly
}
```

This shouldn't be new to you. You do the same with built-in types—although not quite, because when you return an `int`, you don't write `return int(12);`, even though you could.

Therefore, instead of writing

```
return Person{name, age, city};,
```

simply write

```
return {name, age, city};.
```

The compiler knows from the return type of the function that you want to create a `Person`. So it tries to convert the list you provided into a `Person`; the two expressions are almost synonymous. You can choose which style you prefer.

There are cases where you need to specify `Person{...}`—for example, when using the syntax for function declaration available since C++14, `auto create()`. The next listing shows an example.

```
// https://godbolt.org/z/jer44aWs9
auto create(string name, int age, string city) {
    return Person{name, age, city}; // auto requires constructor name
}
auto create2(string name, int age, string city) {
    return {name, age, city};        // ✗ auto with initializer_list does not work
}
```

Listing 12.4 Here, specifying `Person` in the return is necessary.

Here, an `initializer_list` is returned in `create2`. That would be fine if you had specified a return type in the function header into which the `initializer_list` can be automatically converted. But here it says `auto`, and the compiler cannot infer the type from that.

12.3 Methods Instead of Functions

The `print` function in [Listing 12.1](#) requires a `Person` to work; it is in fact specifically designed for `Person` and nothing else. It might even be the case that the `Person` type only really makes sense if you also provide a suitable set of functions that work with your new type.

These functions, along with the data in `struct Person`, belong to the interface of this type. It is a pity that `print` is not already listed in the `Person` type—but you can do exactly this. Combine the functions of a type with the data to form an entity such that neither can exist without the other. Such a function always requires a variable, and a variable without the associated functions is almost useless.¹ A variable that is part of

¹ The static methods that do not require instance variables will be covered later.

the type is called an *instance variable*. A function that is part of the type is called a *method*.

```
// https://godbolt.org/z/5T1Er36P7
#include <string>
#include <iostream>
#include <format>
using std::string; using std::cout; using std::format;
struct Person {
    string name_;
    int age_;
    string city_;
    void print();           // function as a method of the type
};
void Person::print() {     // method name is extended by Person::
    cout << format("{} ({{}}) from {{}}\n",
        name_, age_, city_); // in a method you can directly access fields
}
int main() {
    Person john {"John", 45, "Boston" };
    john.print();          // calling the method for a variable of the type
}
```

Listing 12.5 Methods bundle data and behavior together.

Now `print` is part of the `Person` type and thus has become a *method*.

When one speaks of a *method*, they mean a function within a type. To be distinguished from this are *functions* outside of classes, which you have learned about so far.

Sometimes functions are also called *free functions* to distinguish them from methods. The term *global functions* is also sometimes used, but it is not entirely correct; a function can also be within a namespace and is still free, but not global. Conversely, the term *member function* is used for method.

Technically speaking, methods are also just functions. One of the differences is that the full identifier of a method includes the class name. If you define the method outside the type, then you call it—for example, `Person::print`. Within the method, you can then access all data fields and methods of the class without further qualification: where before you wrote `p.name_`, now `name_` is sufficient.

Strictly speaking, the compiler internally translates this to `this->name_`—and here I come to the second important difference from functions: a method from `Type` always has an implicit `Type* this` parameter,² without it appearing in the method's parameter list. This parameter represents the memory address of the instance on which you call

² More precisely: `Type * const` or `Type const * const`, depending on the case.

the method. this together with the data field you use gives the exact location you mean.

For this, the compiler must use the dereference operator `->` instead of the dot, as described in detail in [Chapter 20](#).

```
struct Person {
    //... rest as before ...
    string greeting();
};
string Person::greeting() {
    return format("Hello {} from {}", this->name_, this->city_);
}
int main() {
    Person anna { "Anna", 33, "Eek" };
    Person nina { "Nina", 22, "Ojo" };
    anna.greeting();
    nina.greeting();
}
```

Listing 12.6 The “greeting()” method uses fields; via “this”, it always refers to the field belonging to the called object.

Within a method, the compiler automatically tries to see if an identifier belongs to `*this`. Thus, you can implement `greeting()` more clearly:

```
string Person::greeting() {
    return format("Hello {} from {}", name_, city_);
}
```

Although the program code during the execution of `greeting()` is the same in both cases, the implicit `this` parameter ensures that different `name_` and `city_` fields are accessed—namely, those belonging to the current object.

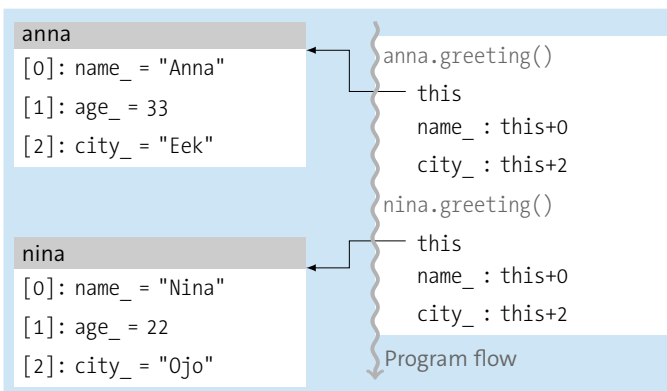


Figure 12.1 Each call to “greeting()” receives a different “this” parameter.

And indeed, with a few small (but important) differences, you can rewrite a method and turn it into a free function by making the implicit `this` parameter explicit.

```
string greeting(Person * const p) { // implicit parameter made explicit
    return format("Hello {} from {}", p->name_, p->city_);
}
```

Listing 12.7 How to separate methods and data from each other.

You must not name the parameter `this` because `this` is a keyword in C++; here, it is called `p`.

There are cases where you also need to use `this->` for attribute access in a method—namely, when two or more identical identifiers conflict. Then you need to be explicit:

```
// https://godbolt.org/z/hEvqWqqc6
int value = 5; // global variable
struct Wrap {
    int value = 3; // data field
    void set(int value) { // parameter
        this->value = value + ::value;
    }
};
```

Here, `this->value` refers to the data field of `Wrap`, `value` refers to the parameter, and `::value` refers to the global variable.

It is a good idea to bundle data and functions together. Whether you should always and consistently do this is a design question. For example, C++ allows the extension of existing functionality through new overloads of free functions, not just by adding methods to an existing data type. The operator<< for `std::ostream` is a good example of this.

12.4 The Better “print”

It pains me every time I present you with something as impractical as `print` as an example of a function. There are several important things to note about `print` as a method or the corresponding free function, simple as it may be. You now know enough about C++ that I can explain the better alternative to you. Take another look at the method:

```
void Person::print() {
    cout << format("{} ({} from {}", name_, age_, city_);
}
```

Here, output is sent to `cout`. What if you want to output a `Person` as an error (to `cerr`)? Would you write a separate function or method for that? The root problem is that you are accessing the `cout` *global variable* in the method. You cannot change this behavior from the outside if, for example, you wanted to write to a file via `ofstream` or to a `stringstream` for testing purposes.

No Global Accesses

Avoid direct access to global variables in functions and methods.

Maybe you want to write automatic tests for `Person::print`. How annoying that you then have to monitor `cout` with your eyes to check if the program ran correctly! I want to suggest that you can make automatic testing (see [Chapter 14](#)) easier, among other things, if you avoid using the global `cout` specifically and global variables in general.

Pass the target of the output as a parameter. `cout` and `cerr` are of type `std::ostream`. And if you want to write to a file, then `ostream` is also suitable for that.

```
void Person::print(std::ostream& os) {
    os << format("{} ({{}} from {})", name_, age_, city_);
}
```

Note that you use the type `ostream&`, which is a reference. You do not want to create a copy of the stream, but rather change its state. Outputting something to a stream is a state change.

Wonderful—having suggested this to you for the future already greatly eases my conscience! Now use the `print` function in the following listing as you like.

```
// https://godbolt.org/z/bz6n3Pjrv
// Excerpt. Person as before
void Person::print(std::ostream& os) {
    os << format("{} ({{}} from {})", name_, age_, city_);
}

int main() {
    Person carl {"Carl", 12, "Toledo"};
    carl.print(cout);           // on the screen
    cout << "\n";
    std::ofstream file {"persons.txt"};
    carl.print(file);          // to a file
    // automatic test:
    std::ostringstream oss{}; // writes to a string
    carl.print(oss);
    if(oss.str() == "Carl (12) from Toledo") {
        cout << "ok\n";
    } else {
```

```

        cout << "Error in Person::print!\n";
        return 1;           // propagate error outward
    }
}

```

Listing 12.8 “print” takes a stream as an argument.

In particular, the use of `ostream` reveals a special aspect: if you want to automatically test whether a printing function works correctly, a `string` is easy to handle and check for the desired result. How would you test that if you had written directly to `cout`? Extracting the output would have become difficult. One of the real advantages of passing the output stream is that the function is easy to test.

12.5 An Output Like Any Other

The next step is actually just cosmetic. To output an object to an `ostream`, there is a commonly used method in practice: overload the `operator<<` global operator for `ostream&` and your type.

```

std::ostream& Person::print(std::ostream& os) {
    return os << format("{} ({} from {})", name_, age_, city_);
}
std::ostream& operator<<(std::ostream& os, Person p) {
    return p.print(os);
}

```

Listing 12.9 You can overload the standard operator for output.

Instead of calling `p.print()`, you can also output the fields directly:

```

std::ostream& operator<<(std::ostream& os, Person p) {
    return os << format("{} ({} from {})", p.name_, p.age_, p.city_);
}

```

Did you notice that the return type is not `void`? By returning the `ostream` passed as a parameter, you can reuse this return value within an expression. You have done this often for `operator<<`. See your new output operator in action in the following listing.

```

// https://godbolt.org/z/a7q5xsWGj
// Excerpt ...
    std::ostream& print(std::ostream& os);
};
std::ostream& Person::print(std::ostream& os) {
    return os << format("{} ({} from {})", name_, age_, city_);
}

```

```
std::ostream& operator<<(std::ostream& os, Person p) {  
    return p.print(os);  
}  
int main() {  
    Person paul {"Paul", 23, "Irvine"};  
    cout << "You are " << paul << ", right?\n";  
}
```

Listing 12.10 The output with << is achieved by overloading a free function.

And this is how you can use `Person` objects in the output just like other data types. As a complete output, you see the following:

You are Paul (23) from Irvine, right?

12.6 Defining Methods Inline

Did you notice that when declaring and later defining the method of the class, I used a technique you already know—the *forward declaration* of the method? First, in the structure, only the *method header* is mentioned, and then later, outside, the actual definition takes place. (Strictly speaking, it is still an aggregate, but it will soon become a class. Everything is still public, without constructors or virtual methods.)

This is also the way you should usually go forward with structures and classes. However, if a method is very short, for example, then you can treat it the same way as normal functions. You can also *define* it right on the spot—within the structure.

```
// https://godbolt.org/z/Y3eW1Kh8v  
#include <string>  
#include <iostream>    // ostream  
#include <format>  
  
using std::string; using std::ostream; using std::format;  
  
struct Person {  
    string name_;  
    int age_;  
    string city_;  
    ostream& print(ostream& os) {        // method defined inline  
        return os << format("{} ({} from {})", name_, age_, city_);  
    }  
};
```

Listing 12.11 Methods can also be defined inline.

However, use this option only when the method is really short.

By writing the method in a class this way, the compiler tries to insert the code of the implementation directly where you call the function: this is called *inlining*. While this potentially makes your code faster (a function call costs a lot of time), it also makes it larger.

If you inline too many functions, your code might become so large that the speed advantage is negated. It is best to inline only methods that are simple or called very, very frequently.

12.7 Separate Implementation and Definition

It is beneficial for the program's clarity if you separate definition and implementation. This even goes so far that you should put the structure definition in a header and the definition in an implementing *.cpp file (see [Figure 12.2](#)).

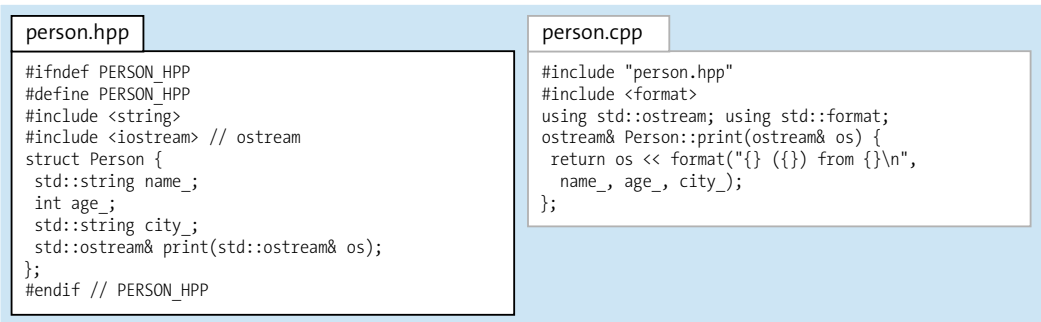


Figure 12.2 Splitting a class into the header and implementation.

If you need `Person` in multiple parts of the program, use `#include "person.hpp"` there to make the data type known to the compiler.

Just in case you need to include `person.hpp` multiple times through various includes, the surrounding lines are there:

```

#ifndef PERSON_HPP
#define PERSON_HPP
// ... actual content
#endif // PERSON_HPP
  
```

These *include guards* prevent `struct Person` from being defined twice if you write `#include "person.hpp"` directly or indirectly twice. It is not *necessary* to use these guards, but it is very common practice.

No “using” in the Header

As I mentioned at the beginning of the book, for the sake of brevity, I do not use `using` quite as you should in practice. In [Figure 12.2](#), you can see it done correctly:

- Do not use global `using` in a header.
- In a `*.cpp` file, you can use `using std::identifier`.
- However, you should not use `using namespace std`; globally, not even in the `*.cpp` file.

12.8 Initialization via Constructor

You are already familiar with the simple `struct Person`:

```
struct Person { string name_; int age_; string city_; };
```

You also know that you can initialize the aggregate `Person` with its three elements in four correct ways:³

```
Person p1 { name, age, city };
Person p2 { name, age };
Person p3 { name };
Person p4 { };
```

Now, if we add the *designated initializers* introduced in C++20, there are seven more possibilities:

```
Person p1 { .name_=name, .age_=age, .city_=city };
Person p2 { .name_=name, .city_=city };           // and 2 more
Person p3 { .age_=age };                         // and 2 more
```

All these variants are valid and initialize all fields. The ones not listed are value-initialized, meaning they are filled with a meaningful, usually null-like value or by a default constructor. But what do you do if you want to enforce that a `city_` is always specified during initialization? Or if you want to offer the possibility to omit `name_` instead of `city_` without having to specify the designated initializer?

For this purpose, there are constructors. A *constructor* is a special method of the structure, one specifically designed to initialize the object. You write a constructor *almost* like a normal method, with the following differences:

³ Here, I only consider the uniform initialization with curly brackets `{...}`. Not mentioned are the constructions with parentheses or without parentheses, as well as copy and move. I will address these forms later.

- The name of the constructor is always the name of the structure—here, `Person(...)`.
- A constructor has no return value or type, not even `void`.
- Before the constructor body, you can (and should) initialize the fields of the class with `: field{value}, ...`

You must declare the constructor like methods within the `struct`:

```
struct Person {
    string name_;
    int age_;
    string city_;
    Person();           // declare constructor
};
Person::Person()
    : name_{"no name"} // initialization value for name_
    , age_{-1}         // initialization value for age_
    , city_{"no city"} // initialization value for city_
{ }                  // empty function body
```

Listing 12.12 How to define a constructor.

If you now initialize a `Person p{}`, the value initialization will no longer be executed; instead, this constructor will be. Before the (here empty) function body is entered, all variables are initialized with the specified values. Instead of curly brackets `{...}`, you can also use parentheses `(...)`. This makes no difference here—except that the curly brackets might lead you to prefer this form elsewhere. You can also choose a different formatting than was shown earlier—for example:

```
Person::Person()
    : name_("no name"), age_(-1), city_("no city")
{ }
```

Initializations that you do not specify here will be performed as if you did not initialize a variable of the corresponding type. That means that built-in data types remain undefined and classes are value-initialized.

Self-Initialize or Rely on Rules?

I advise you to initialize all data fields. If you stumble upon a forgotten initialization in someone else's code, you should be allowed to add the missing initialization without shame. The rules for what to initialize when and how, if the mention is omitted, are too complicated, and even gurus like Scott Meyers do not want to remember the rules. The main reason for this is, of course, not a lack of brain capacity, but that it is in the nature of the matter that the definition of the data field is far removed from its initialization. And looking up the declaration every time during initialization to decide whether the

initialization should be mentioned or not is unbearable. In conclusion: always self-initialize.

Listing all data fields in every constructor is a lot of work and creates suboptimal code duplication. Fortunately, there are alternatives like default member initializers and constructor delegation that make it easier for you. But we will come to those in the next section.

So, you get the option to fill the structure with other values using `Person p{}`; . However, you can no longer specify one, two, or three arguments.

A Constructor Makes an Aggregate into a Structure

With a custom constructor, there is no more aggregate or value initialization for this type. You must then use one of the constructors for initialization. A different number of parameters than is defined by the constructor(s) is no longer possible for initialization.

To distinguish, this type is then called a *structure*. The term *aggregate* also applies to other language elements, such as the C-array.

You will later get to know other elements that make an aggregate into a structure or class—namely, virtual methods and nonpublic data fields.

Like a normal function, you can overload the constructor with different parameter combinations to also allow initialization with multiple parameter combinations.

```
// https://godbolt.org/z/zndbv5nK
#include <string>
#include <string_view>
using std::string; using svview = std::string_view;

struct Person {
    string name_;
    int age_;
    string city_;
    Person(); // constructor without arguments
    Person(svview n, int a, svview c); // constructor with three arguments
    Person(svview n, int a); // constructor with two arguments
    Person(svview n); // constructor with one argument
};

Person::Person()
    : name_{"no name"}, age_{-1}, city_{"no city"} { }
Person::Person(svview n, int a, svview c)
    : name_{n}, age_{a}, city_{c} { }
```

```

Person::Person(sview n, int a)
    : name_{n}, age_{a}, city_{"no city"} { }
Person::Person(sview n)
    : name_{n}, age_{-1}, city_{"no city"} { }

```

Listing 12.13 Multiple constructors are also possible.

This allows flexibility in initialization, but it is not ideal: it contains a lot of code duplication and thus sources of errors in later code maintenance. You can remedy this situation for `Person` in three ways:

- With member default values at declaration
- With constructor delegation
- With default values for constructor parameters

You can also see that it is good to declare a read-only `string` parameter as `string_view` since C++17. Alternatively, you could use `string` as a value parameter or `const string&` as a reference parameter.

12.8.1 Member Default Values in Declaration

You can provide a member variable in the `struct` with an `=` and a value. If you then omit this variable in the list of initialization elements in the constructor, this value will be used instead. The preceding example is therefore equivalent to the following listing.

```

// https://godbolt.org/z/Wo8zaq9WW
#include <string>
#include <string_view>
using std::string; using sview = std::string_view;
struct Person {
    string name_ = "no name";
    int age_ = -1;
    string city_ = "no city";
    Person() {}
    Person(sview n, int a, sview c)
        : name_{n}, age_{a}, city_{c} { }
    Person(sview n, int a)
        : name_{n}, age_{a} { }
    Person(sview n)
        : name_{n} { }
};

```

Listing 12.14 Member variables can be equipped with default values.

The values mentioned in the declaration are only used if the initialization in the constructor is actually omitted. This means, in particular, that no double initialization takes place.

12.8.2 Constructor Delegation

You can call another constructor of this class first in the list of initializers, thus delegating the initialization of the values to it. For this, implement a constructor with the maximum number of arguments and then delegate the initialization of the member variables from all other constructors to it (see [Listing 12.15](#)).

With this variant, you will not get rid of all code duplications for `Person`. This is useful if something is also within the—here empty—constructor body, because the code block of the constructor to which you delegate is also executed. Only when this is finished, the code that may be in the delegating constructor is executed. Using delegation like this reduces the code you have to write in each constructor.

```
// https://godbolt.org/z/8h93f64Yj
#include <string>
#include <string_view>
using std::string; using sview = std::string_view;
struct Person {
    string name_;
    int age_;
    string city_;

    Person(sview n, int a, sview c)           // delegated constructor
        : name_(n), age_(a), city_(c) { }    // ... implemented
    Person() : Person{"no name", -1, "no city"} { } // delegating
    Person(sview n, int a) : Person{n, a, "no city"} { } // delegating
    Person(sview n) : Person{n, -1, "no city"} { } // delegating
};
```

Listing 12.15 A constructor can pass part of the initialization to another constructor.

Leave the Body of the Delegating Constructor Empty

Be careful if an exception is thrown in the body of the constructor that has previously delegated elsewhere. The object is considered created after the delegated constructor runs. This means that it will eventually be cleaned up and the destructor will be called. This could then come unexpectedly as without delegation, an object is considered *not* created if an exception leaves the constructor. You will learn about the effects of the destructor call in [Chapter 16](#).

To avoid difficulties, you should leave the body of the delegating constructor empty. This way, no exception can be thrown there.

12.8.3 Default Values for Constructor Parameters

You can write a constructor that flexibly takes many arguments with default parameters. How was already described in [Chapter 7, Section 7.9](#). The following listing offers a short example.

```
// https://godbolt.org/z/Gfrd5cceG
#include <string>
#include <string_view>
using std::string; using svview = std::string_view;

struct Person {
    string name_;
    int age_;
    string city_;

    Person(svview n = "N.N.", int a = 18, svview c = "Berlin")
        : name_(n), age_(a), city_(c) { }
};
```

Listing 12.16 A constructor can also be overloaded with default parameters.

This single constructor definition then behaves roughly as if you had defined the following four overloads:

```
Person(svview n, int a, svview c)
    : name_(n), age_(a), city_(c) { }
Person(svview n = "N.N.", int a = 18)
    : name_(n), age_(a), city_("Berlin") { }
Person(svview n = "N.N.")
    : name_(n), age_(18), city_("Berlin") { }
Person()
    : name_("N.N."), age_(18), city_("Berlin") { }
```

Note that the constructor that provides default values for all its arguments is also used as the *default constructor*. This means that the compiler calls it whenever an instance needs to be created without arguments. This is the case, for example, with `vector`.

```
// https://godbolt.org/z/1fE15Kf39
#include <vector>
#include <string>
#include <string_view>
#include <iostream>
using std::string; using std::cout; using svview = std::string_view;
```

```
struct Person {
    string name_;
    int age_;
    string city_;
    // acts as the default constructor:
    Person(sview n = "N.N.", int a = 18, sview c = "Berlin")
        : name_(n), age_(a), city_(c) { }
};

int main() {
    std::vector<Person> people{};    // initially empty
    people.resize(5);               // expand to five 'empty' people
    cout << people[3].city_ << "\n"; // Output: Berlin
}
```

Listing 12.17 A constructor with all preset arguments becomes the default constructor.

Here, `resize(5)` causes the vector to create five new `Person` instances. Without further specifications, the vector uses the default constructor, which in this case is the one we provided, with all its arguments preset with default values.

So, when the standard (or this book) refers to a *default constructor*, it can indeed be one that receives multiple parameters—so long as they are all preset.

12.8.4 Do Not Call the “init” Method in the Constructor

Do not be tempted to write a normal method that handles initialization, perhaps `init()`, and then call it in the constructor body. By then, it is too late to initialize member variables; you can only *assign* them values (and that would be redundant work).

There is not much to object to about an initialization method like `init()` itself. However, you are not *initializing* in the C++ language sense here, but assigning new values to all member variables. A better name in [Listing 12.18](#) might have been `assign()` or `set()`.

But it is wrong to use this as a substitute for the constructor initialization list—because it is not one. The member variables have already been initialized, even though you omitted the colon list `: ...`—just value-initialized, if possible. Thus, `init()` can only make assignments, and work has been done twice: first the initialization was done, then the value was overwritten with the assignment.

Under no circumstances should you proceed this way if your data type has methods that are marked with `virtual`; see [Chapter 15](#). *Remember:* You *must not* call a *virtual method* from a constructor. Your program will likely crash.

```
// https://godbolt.org/z/dzjaf6nbb
#include <string>
#include <string_view>
using std::string; using svview = std::string_view;

struct Person {
    string name_;
    int age_;
    string city_;
    Person(svview n, int a, svview c)
    {
        // ✗ Initialization list missing
        init(n, a, c); // ✗ questionable »initialization call«
    }
    void init(svview n, int a, svview c) {
        name_ = n; age_ = a; city_ = c;
    }
};
```

Listing 12.18 Do not call an initializing method in the constructor body.

So feel free to use such an `init` method, but only when the object is already completely finished and the constructor code has already been exited. Sometimes you need this approach to initialize your objects in two phases: if the objects need each other, first construct all of them and then call the necessary `init()` methods.

12.8.5 Exceptions in the Constructor

Exceptions have a special significance within constructors. They prevent the object from being created. You can see the exact effects in [Chapter 16](#).

12.9 Struct or Class?

A type that contains a constructor is no longer an aggregate; it is a *structure*. This term comes from the fact that you introduce the type declaration with `struct`.

You will sometimes hear the term *class* instead—and that is also correct, because instead of `struct ...` you can just as well write `class ...`. There is only a tiny difference in meaning between `struct` and `class` in C++:

- **struct**
A `struct` initially has public access rights.
- **class**
A `class` starts with private access rights.

Having *public access rights* means that you can access the internals of the type from the outside. You saw this in the implementation of `operator<<`:

```
ostream& operator<<(ostream& os, Person p) {  
    return p.print(os);  
}
```

The `print` method of the `Person` type can be accessed by `operator<<` as a free function (from the outside) because it is *public*. The data fields are also public, so you could have written the following listing instead.

```
ostream& operator<<(ostream& os, Person p) {  
    return os << format("{} ({} from {})", p.name_, p.age_, p.city_);  
}
```

Listing 12.19 External access to the data fields of a type.

Accessing the internals of the type in this way completely contradicts what was actually intended with the introduction of methods: that it is the type itself that provides and maintains the functionality. For this purpose, you can—and should—restrict public access to parts of a type. Separate a specific range using `private:`. From the point of declaration, methods and data are then protected from external access. You achieve the opposite with `public:`—that is, everything behind this keyword is public.

With the changes in [Listing 12.19](#), it is no longer possible to directly access `name_` and the other data.

```
// https://godbolt.org/z/4MzxaeVka  
#include <string>  
#include <string_view>  
using std::string; using std::string_view;  
  
struct Person {  
private: // everything from here cannot be used externally  
    string name_;  
    int age_;  
    string city_;  
  
public: // everything from here can be used externally  
    Person(string_view n, int a, string_view c)  
        : name_{n}, age_{a}, city_{c} { }  
    void print();  
};
```

Listing 12.20 Divide a type into multiple sections with “public” and “private”.

12.9.1 Encapsulation

The concept of protecting certain things from external access is called *encapsulation*. This mainly means packing the data itself into the private section and allowing access only through methods, thereby controlling it.

In fact, there are other forms of encapsulation; `public` and `private` sections of a class are just one aspect. In C++, the `private` parts of a class must also be listed. Thus, programmers who can see the class definition can also see the names and types of the private data. It can be quite useful to further pursue the concept of *information hiding* beyond encapsulation. For more information, check [Chapter 11](#) about modularization, where the Pimpl pattern is used to hide an implementation.

12.9.2 “public” and “private”, Struct and Class

Let's return to the difference between `class` and `struct`: For the compiler, the only difference is that a `struct` implicitly starts with `public`: and a `class` starts with an implicit `private`:

So you could have written [Listing 12.20](#) as in the following listing, and it would have meant exactly the same thing.

```
// https://godbolt.org/z/MYzsqqdo
#include <string>
#include <string_view>
using std::string; using std::string_view;

class Person { // a class starts with private visibility
    string name_;
    int age_;
    string city_;
public:      // everything from here can be used externally
    Person(string_view n, int a, string_view c)
        : name_{n}, age_{a}, city_{c} { }
    void print();
};
```

Listing 12.21 A “class” starts with private visibility.

Semantically, in C++, a class and a structure are the same. However, the difference arises from the default visibility you get depending on whether you start your type with `struct` or `class`.

12.9.3 Data with “struct”, Behavior with “class”

Apart from this technical distinction, I recommend that you still make a certain distinction between structure and class in your mind and that you also highlight this by using `struct` and `class`:

- Use `struct` for those types that are primarily holders of data and have only a little behavior through methods.
- Use `class` when you consider the actual data as an implementation detail, and it is the behavior that you want to expose.

12.9.4 Initialization of Types with Private Data

You have already learned that if you add at least one constructor to your data type, you can no longer value-initialize. The same applies if you pack data into the `private` area of your type.

```
// https://godbolt.org/z/Pr76Y55P4
class Rect {
    int area_; // private data
public:
    int x_, y_;
    void set(int x, int y) { x_=x; y_=y; area_=x_*y_; }
    int calc() { return area_; }
};
```

Listing 12.22 Parts of the data are private.

Because you are no longer allowed to directly access the `area_` member variable from the outside, you can no longer simply fill it with curly brackets. You cannot write `Rect r{1,2};` or `Rect s{6,2,3};`. With `area_` in the private section of the class, value initialization for `Rect` is impossible.

You definitely need a constructor here. Never accept that something lies around uninitialized after definition! Fortunately, the compiler sees it the same way, and therefore *generates* a constructor in these cases—specifically, one without parameters. This allows you to at least use `Rect t{};` (and you should).

And what does this generated constructor do? Effectively, it initializes your object with zeros or, in the case of member variables of more complex types, with an appropriate equivalent. This is called *zero initialization*. (In fact, your object is first zero-initialized, and then the generated constructor is called, which does nothing; the effect is the same here.)

You have to get through a lot of automation first. Think of the readers of your program and provide a constructor, even if it's just the parameterless constructor. On the other hand, if the constructor only does what the compiler would generate anyway, you can

request its creation with `= default`. At least then you've made it explicit, and the readers can see directly: "Oh yes, initialization without parameters."

```
// https://godbolt.org/z/PaG9nTs7s
class Rect {
    int area_;           // private data
public:
    int x_, y_;
    void set(int x, int y) { x_=x; y_=y; area_=x_*y_; }
    int calc() { return area_; }
    Rect() = default;    // let the compiler generate a constructor
};

class Pow {
    int result_;         // private data; holds 'base' raised to 'exp'
public:
    int base_, exp_;
    void set(int b, int e) { /* ... */ }
    int calc() { return result_; }
    Pow() : result_{1}, base_{1}, exp_{1} {} // initialize sensibly
};
```

Listing 12.23 With “`= default`”, you let the compiler generate code.

While `Rect() = default`; only requests zero-initialization, which the compiler would have done anyway, the `Pow()` constructor actually does something. Zero-initialization for `base_` and `exp_` would mean 0^0 , which is mathematically undefined, so the constructor instead sets 1 for both. Thus, `result_` must also be set to 1, because $1^1 = 1$.

Automatically Generated Default Constructor

The compiler generates a default constructor (the one without arguments) only if you do not define any custom constructors.

So, for example, if you add `Rect(int x, int y)`, the compiler will not generate a `Rect()` constructor. The `Rect a{}; initialization will no longer work, but Rect b{4,3}; will. If you still want it, you can declare it additionally with Rect() = default and let the compiler define it.`

12.10 Interim Recap

We’ve covered a lot of ground so far in this chapter. Before we dive deeper into how to make usage of your own data types, let’s take a breather and recall the key lessons learned so far:

- Use `struct` to create new types.
- Global functions can operate on parameters of the new type and return the new type as a result.
- Methods keep data and functions together.
- Avoid using global variables directly in functions and methods.
- You can define `operator<<(ostream&,...)` and use it to output your type.
- Separate the declaration and definition of methods by splitting them into header and implementation files.
- Divide your type into private and public sections using `public` and `private`.
- Use `struct` or `class` to clarify your intention for the type.

12.11 Using Custom Data Types

You now know how to create custom data types, how to initialize them, and how to equip them with methods. I will now show you a detailed example of how to use such custom types. Here you will see the following, among other things:

- How to pass and return a custom data type as a parameter
- How the type can be automatically converted
- How to output it and use it for calculations
- How it protects you from errors by providing type safety

Consider a date calculation. When dealing with a date with *year*, *month*, and *day*, there is always the risk of swapping the month and day, and so on. The ISO date 2024-02-05 is written in the US as 2/5/24, while it's 5.2.24 in Germany, for example. If you have a `void printDate(int y, int m, int d);` function and call it somewhere with `printDate(28, 2, 2024)`, then that is obviously wrong.

This happens if you no longer remember the exact meaning of the parameters. Then the program will probably not do what you intended, because you should have written `printDate(2024, 2, 28)`. And nothing but the names of the parameters can help you use the function correctly.

The following example is safer. If you introduce different types for the various time units, you protect `Date` from incorrect usage. You are already quite close to this goal with this example. But beware: you have not yet fully achieved it, as you will see in the discussion. Examine the following complete listing first.

```
// https://godbolt.org/z/aGd56xzsT
#include <string>           // string, stoi
#include <iostream>         // cin, cout, ostream
#include <format>
using std::ostream; using std::format;
```

```

class Year { /* Helper types for safe date */
    int value_; // e.g., 2024
public:
    Year(int v) : value_{v} {}
    int value() { return value_; }
};

class Month {
    int value_; // 1..12
public:
    Month(int v) : value_{v} {}
    int value() { return value_; }
};

class Day {
    int value_; // 1..31
public:
    Day(int v) : value_{v} {}
    int value() { return value_; }
};

/* type-safe constructing date */
class Date {
    Year year_;
    Month month_ = 1;
    Day day_ = 1;
public:
    Date(int y) : year_{y}{} // 1-argument constructor
                        // sets 1st January of the specified year

    Date(Year y, Month m, Day d) // 3-argument constructor
        : year_{y}, month_{m}, day_{d}{}
        ostream& print(ostream& os); // e.g., 2024-04-20
};

ostream& Date::print(ostream& os) { // e.g., 2024-04-20
    return os << format("{}-{:02}-{:02}",
        year_.value(), month_.value(), day_.value());
}

ostream& operator<<(ostream& os, Date d) {
    return d.print(os);
}

// http://codegolf.stackexchange.com/a/11146/1405, user Fors, 2014-02-25
Date easter(Year year) {
    const int y = year.value();
    int a = y/100*1483 - y/400*2225 + 2613;
    int b = (y%19*3510 + a/25*319)/330%29;
    b = 148 - b - (y*5/4 + a - b)%7;
    return Date{Year{y}, Month{b/31}, Day{b%31 + 1}}; // Create date
}

```

```
int main(int argc, const char *argv[] ) {
    /* Input */
    int number {};
    if(argc > 1) {
        number = std::stoi(argv[1]);
    } else {
        std::cout << "Year? "; std::cin >> number;
    }
    /* Calculation */
    Date date = easter(number); // implicit conversion to Year
    /* Output */
    std::cout << "Easter: " << date << "\n";
}
```

Listing 12.24 Custom data types can protect against errors. This is the first step toward that.

When executed, the program asks for a year like 2024 and then outputs the date of Easter Sunday for that year. You can also call the program from the command line with the year:

```
$ ./33-easter.x 2024
Easter: 2024-03-31
```

The function for calculating Easter is a variation of Carl Friedrich Gauss's algorithm. He was the first to devise a simple calculation rule without large tables. In his time, this was one of the most important calculations ever. I will not go into the details of this calculation here.⁴

One line is particularly interesting upon closer inspection:

```
Date date = easter(number);
```

Here, the `int number` is automatically converted into a `Year` for the parameter of `easter`. Why this happens and how it works is shown in [Section 12.11.3](#), under “Implicit Type Conversion.” There you will also see whether this is a good idea or not.

12.11.1 Using Classes as Values

When working with your custom data types, you should treat them no differently than those from the standard library—and actually no differently than the built-in types. When using `classes` and `structs`, first consider whether there is any reason not to use them as if they were an `int` or similar. Only if there is a good reason not to do so should you handle an individual case differently.

⁴ See *Date of Easter*, https://en.wikipedia.org/wiki/Date_of_Easter, [2024-05-19]

This means that you should also treat variables of your class as simple *values*, and so the following applies:

- You can assign a completely new value to a variable of your type, instead of changing its internal state.
- Prefer passing a variable by value (*call-by-value*). Only when you are sure should you use references (or pointers).
- Do not hesitate to return your type as a result.

Value Parameters

In the example, the functions receive their own data types as value parameters:

```
ostream& operator<<(ostream& os, Date d);
Date easter(Year year);
```

Both functions only read their parameters and do not want to change them. Two ways to pass parameters in this case are by *value* and by *constant reference*.

Passing by value is the safest way to protect your program as it grows from getting confused in a mess of references. Pass a reference, and if the original object no longer exists outside, then the program may crash as soon as the reference is used in the function.

The disadvantage is that the variable has to be copied for this. This costs time and memory during program execution. However, you should still prefer this option to protect yourself from hard-to-find errors in a larger project.

An alternative to passing by value is the *constant reference*. It looks like this in this case:

```
ostream& operator<<(ostream& os, const Date& d) {
    return d.print(os); // ⚡ d is const
}
```

But this now results in an error: `d` must not be changed because that is exactly what you said with `const Date&`. However, `d.print...` *could* change `d`; at least, that's how it looks to the compiler. To remedy this, you need to tell the compiler that `print` does not change the object. To do this, add a `const` after the method declaration:

```
ostream& print(ostream& os) const;
```

Now the `d.print(os)` call works—and additionally, you are now also *const-correct*. I go into all these aspects of `const` in detail in [Chapter 13](#).

Returns

If you want to return your new type, just do it. Normally, you do not return a reference—and if you do, it is at most one that you received as a parameter, like `os` in the following:

```
ostream& Date::print(ostream& os) {  
    return os << format("{}-{:02}-{:02}",  
        year_.value(), month_.value(), day_.value());  
}
```

Never—and I mean *never*!—return a reference to an object created (nonstatically) within the function or method:

```
Date& firstJanuaryOf(int y) { // ⚡ returning a nonconst reference is a red flag  
    Date result{Year{y}, Month{1}, Day{1}};  
    return result;           // ⚡ reference to local variable  
}
```

This immediately goes wrong: `result` is created in this function and will be destroyed upon exit. If you pass the reference to the internal object outside the function, then the reference points to the destroyed object. This is simply wrong.

Therefore, just pass the newly created object outward as a value:

```
Date firstJanuaryOf(int y) {  
    Date result{Year{y}, Month{1}, Day{1}};  
    return result;  
}
```

Performance when Returning

If you are worried that there might be extra copying here, I can reassure you: the compiler can avoid a copy in most cases (*copy elision*). You are on the safe side performance-wise if you have exactly one `return` in the function or if there is only one place in the function where your return value is created. And even if neither is true, the compiler can often still avoid a complete copy.

```
// https://godbolt.org/z/T7afh4vxz  
#include <vector>  
std::vector<int> createData(unsigned size) {  
    std::vector<int> result{};  
    for(int idx=0; idx<size; ++idx) {  
        result.push_back(idx);  
    }  
    return result;  
}
```

Listing 12.25 If all “return” statements return the same variable, the compiler can always avoid a copy.

The normal procedure would be as follows, assuming the return type is R :

- Call the function.
- Locally create a new instance f of R .
- Populate f in the function.
- Return to the call site.
- Create a new instance a of R at the call site.
- Initialize a from f using the copy constructor.
- Clean up f .

However, if certain rules apply, the compiler can simplify this:

- Create a new instance a of R at the call site.
- Call the function.
- Populate the instance a created at the call site in the function.
- Return to the call site.

In addition to saving the copy of the object, you also save a construction and a destruction.

The rules state that the caller and the function must behave as if f and a were the same object, thereby saving creation, copying, and deletion. How the compiler does this is in its implementation detail. For example, the caller could already determine the address of a where the function is to create the object and pass it as a hidden parameter.

12.11.2 Using Constructors

In the `easter()` function, the following line is particularly interesting:

```
return Date{Year{y}, Month{b/31}, Day{b%31 + 1}};
```

Here, a `Date` is created by calling the constructor with three arguments:

```
Date(Year y, Month m, Day d);
```

The arguments are three fresh variables of different types, explicitly called with their respective one-argument constructors: `Year{y}` creates a temporary variable of type `Year` from the `int y`, and so on.

Because the three constructor arguments are of different types, it cannot happen that you accidentally use

```
return Date{Day{b%31 + 1}, Month{b/31}, Year{y}};.
```


The constructor does not match for that. You have achieved safety here. In many APIs that provide a type for a date, the constructor simply takes three `int` values, and the detection of accidental swapping of arguments is lost.

Let the compiler help you to avoid errors.

12.11.3 Type Conversions

There are several ways to convert a value to another type.

Implicit Type Conversion

Where the function `easter()` is used, you can see a great feature. You know that `number` is of type `int`. And yet it says the following:

```
Date date = easter(number);
```

Although the function `easter` takes an argument of type `Year`:

```
Date easter(Year year);
```

You do not need to take a detour to first create a `Year`:

```
Year y{number};  
Date date = easter(y);  
// or  
Date date = easter(Year{number});
```

The compiler sees that it is possible to convert an `int` to a `Year` here. This is called *implicit type conversion*.

Type Conversion via Constructor

A constructor with one argument allows the compiler to perform an implicit type conversion: from the type of the argument to the type to which the constructor belongs.

When the compiler sees `easter(number)`, it realizes that `number` is an `int`, but `easter` requires a `Year`. The `Year(int)` constructor allows exactly this conversion—and thus the compiler creates a temporary variable itself, as if you had called `easter(Year{number})`.

Explicit Type Conversion

I already mentioned that this example has a catch. The implicit type conversion presents you with a pitfall, and a big one at that.

Because what applies to functions with one argument like `easter(Year)` also applies to those with three arguments.

Try inserting `int` values instead of the types for the different time units into the constructor of `Date`:

```
/* return Date{Year{y}, Month{b/31}, Day{b%31 + 1}}; */
return Date{y, b/31, b%31 + 1};
```

Oh dear, that works! The `Date` constructor (similar to a function in this sense) handles three `int` values just fine—logically so as all three calendrical types have constructors with one `int` as an argument and now allow the compiler to perform the implicit conversion:

```
Year::Year(int v);
Month::Month(int v);
Day::Day(int v);
```

You have already seen that the implicit conversion by the compiler can sometimes be useful. But in many cases, like here, it can be dangerous. Now it deprives us of the protection against swapping the arguments. Because even `Date{b%31 + 1, b/31, y}` is accepted by the compiler, but that is wrong.

Prevent Conversion with “explicit”

The solution is the `explicit` keyword. If you add this to the declaration of a single-argument constructor, then the compiler will no longer choose it for implicit type conversion. In that case, you *must* convert an `int y` to a `Year` by writing `Year{y}`.

The remaining lines of the types remain unchanged, but the declaration of the constructors now looks as shown in the following listing.

```
// only excerpts
class Year {
    explicit Year(int v) : value_{v} {}
};
class Month {
    explicit Month(int v) : value_{v} {}
};
class Day {
    explicit Day(int v) : value_{v} {}
};
class Date {
    explicit Date(int y) : year_{y} {}
};
```

Listing 12.26 With “explicit”, you prevent automatic type conversion.

For a constructor with more than one argument, `explicit` makes no sense. After all, multiple types cannot be converted into another type simultaneously.

The disadvantage is that you now also have to forgo implicit conversion when calling `ostream` and instead make it explicit. However, this additional effort is worth it:

```
Date date = easter(Year{number});
```

Tip

Normally, you should mark constructors with exactly one argument with the `explicit` keyword.

12.11.4 Encapsulate and Decapsulate

The three `Year`, `Month`, and `Day` types essentially do nothing more than *encapsulate* a value. By encapsulating the actual `int value_` value, you have created a unique type that hides the inner value. But of course, you need to somehow get the real value in and out.

I have already described the way in. It goes through the constructor, and `value_` can no longer be changed afterward. If you want a new `value_`, then you also create a new surrounding capsule—for example, `Year a{2023};` and then `a = Year{2024};`

The way out is implemented in the example as follows:

```
int Year::value() { return value_; }
```

This can be seen as a disadvantage: every time you want to get the actual value of `Year`, you have to use `value()`. The `print()` method makes extensive use of this:

```
ostream& Date::print(ostream& os) {
    return os << format("{}-{:02}-{:02}",
        year_.value(), month_.value(), day_.value());
}
```

I would like to show you two ways to do this differently—which are even better under certain circumstances.

Decapsulation

If you want to partially expose the inner values, feel free to use an access function like `int value()`. However, be aware that this requires users of the class to be somewhat familiar with the class's otherwise hidden (because they're private) details. With a method like the following, you are to some extent giving up encapsulation:

```
int Year::value() { return value_; }
```

You expose the `int` type, which is the type of the actually private `value_` variable. This does not always have to be a design exclusion criterion; with the small classes, you

mainly want to achieve type safety for the constructor of `Date`. The alternative, working directly with three `int` values, is, as mentioned at the beginning, not type-safe.

So if you accept a method like `int Year::value()`, you might be bothered that you can't just manipulate `Year` like an `int`. Always writing `year.value()` is more cumbersome than just writing `year`. Because type conversion toward your own type is possible with the help of single-argument constructors, it stands to reason that there is also a way in C++ to convert your own type *into* another type.

The way to convert your type to an `int` is called operator `int()`. Here you can see the complete `Year` class:

```
class Year {
    int value_;
public:
    explicit Year(int v) : value_{v} {}
    int value() { return value_; }
    operator int() { return value(); }
};
```

Now you can use `Year` variables where an `int` was allowed:

```
Year year{2024};
cout << year;           // output like an int
int number = year;      // assign like an int
Year future { year + 10 }; // year to int and back again
```

The last line does the most. In `year + 10`, the operator `+` is applied to two operands of type `Year` and `int`. However, there is no `+` that can directly combine a `Year` and an `int`. The compiler, however, finds out that with operator `Year::int()`, a conversion from `Year` to `int` is possible, and thus a way exists to apply `+` to `int` and `int`. Therefore, `year` is converted to the 2024 `int` and `+10` is executed. The result, 2034, is, of course, again an `int`. This is then the parameter for the constructor call with `Year future{2034}`.

Use Conversion Methods Sparingly

As tempting as it may be to save typing work with `value()`, by adding a new conversion operator, you open a new path for the compiler to apply its automatisms, which can quickly become confusing if used excessively. Sometimes you will wonder how the compiler found *this* path for conversion. This will especially be the case if you also have non-explicit constructors.

You should use conversion methods sparingly. If at all, prefer single-argument constructors. And only if that is not enough should you allow conversion, and only within a small, closely collaborating family of classes. You should particularly avoid conversions to ubiquitous types such as `int`.

Total Encapsulation

Often you should go the other way. If your data type does more than just hide a single value, then expose as little of the internals as possible. So avoid methods like `int Year::value()`.

Nevertheless, there are of course operations that you want to perform. Without operations, a type makes no sense. So far, you have “cheated” by relying on the operations that `int` provides: input, output, calculation, and so on.

If you now do not want to reveal the internals of your class, you must provide the necessary operations for your data type. This can be a lot of work, but it allows for a strict separation of *interface* and *implementation*.

Take `Year`, for example. So far, you have used the `int value()` method to use the actually encapsulated `int` value when you wanted to perform calculations with `Year`. So you have used the operations actually intended for `int` for `Year`. In the Easter example, `Year` was only about the *output*, implemented through `operator<<`.

For other purposes, it might be useful to change the year. This operation is named *advance* with the following method signature:

```
Year& Year::advance(const Year &difference);
```

Note that the current `value_` of the instance is to be changed here. This then allows the following, for example:

```
Year year{2024};
year.advance(Year{1});
cout << year; // Output: 2025
```

It is also useful to be able to check in an `if` statement whether two years are equal. And if you can no longer rely on the `int` value via `value()`, then you need a method for that:

```
bool Year::equals(const Year& other);
// allows:
Year year{2024};
if( year.equals( Year{2020}.advance(Year{4}) ) ) {
    //...
}
```

In this example, you can see why you need operations on your own data type, whose internal values you have completely encapsulated. For the mentioned functions, it would look like the following listing for `Year`.

```
// https://godbolt.org/z/EPz4Kd6jP
class Year {
    int value_;
```

```

public:
    explicit Year(int v) : value_{v} {}
    std::ostream& print(std::ostream& os) const;
    Year& advance(const Year& other);
    bool equals(const Year& other) const;
    bool less_than(const Year& other) const;
};
std::ostream& Year::print(std::ostream& os) const {
    return os << value_;
}
std::ostream& operator<<(std::ostream& os, const Year& year) {
    return year.print(os);
}
Year& Year::advance(const Year& other) {
    value_ += other.value_;
    return *this;
}
bool Year::equals(const Year& other) const {
    return value_ == other.value_;
}
bool Year::less_than(const Year& other) const {
    return value_ < other.value_;
}

```

Listing 12.27 “Year” no longer has “value()” and requires other methods.

The same operations make sense for Month and Day as well. However, you need to think about the *design*: What happens with calculations outside the valid range, such as `Month{12}.advance(Month{1})`? And what about Date? What should happen if you move one day forward from 2024-12-31?

You have some technical questions to clarify, where sometimes there is no right or wrong, but you need to decide on a clear mode of implementation. Which one depends heavily on the requirements.

A quite reasonable requirement could be that `Date::advance()` always returns a valid date. To achieve this, a `Date::normalize()` utility function could help: This checks the partial Year, Month, and Day data for plausibility and makes corrections if necessary. This implies that the partial data itself should not be corrected, as it may need the context of the entire date.

Fluent Programming

I recommended that `advance()` should return the current object with `*this`. In addition, consider the following listing.

```
Year year{2024};
year.advance(Year{1}).advance(Year{3});
cout << year; // Output: 2028
```

Listing 12.28 If “advance” returns the object itself, then you can call another method afterward.

This is similar to the approach of `operator<<(ostream& os,...)`. In this, you end the implementation with `return os`. This allows you to chain multiple instances of `<<` together in one expression, as you have often seen in this book.

This way of designing a programming interface is called *Fluent API*. By returning the object from each function as a reference, you can directly chain the next method call. This results in very readable source code, especially for large, complex classes with many methods. You can even use this concept more intensively and let multiple classes collaborate in this way.

[Listing 12.29](#) shows a sketch as an example to give you an idea of what it might mean. As you can see, this representation closely resembles the structure of the HTML page that is to be generated here. It can be advantageous for this to be reflected in the program code.

```
Page page = Html().body()
    .h1("Heading")
    .table().border(0)
    .tr()
        .td().css("head").text("Dog Breed").end()
        .td().text("Poodle").end()
    .end()
.end()
.toPage();
```

Listing 12.29 A fluent programming interface sometimes allows for clear code.

Marking Methods with “const”

Did you notice that I wrote `const` after the method signatures of `print`, `equals`, and `less_than`? This is necessary to indicate that this method does not change the value of the current instance. This makes your program *const-correct*, which I discuss in detail in [Chapter 13, Section 13.10](#).

12.11.5 Give Types a Local Name

Let's go back to a version of `Year` in which the `value()` method still exists. This can be a sensible design decision.

Sometimes the type used in a class belongs to your *interface*. This means that users need to know the type and use it correctly to properly utilize the class.

For example, users need to initialize the constructor of `Year` with an `int`. They need to know and use this type to avoid other implicit type conversions by the compiler that produce the required `int`. In addition, the return value of `value()` intentionally has the same type. So an expression where the return value is used must also have a matching type—or you must accept another conversion.

```
// https://godbolt.org/z/hbsbMsGr1
class Year {
    int value_; // actually internally used type
public:
    explicit Year(int v) : value_{v} {} // type becomes part of the interface
    int value() { return value_; } // also in the return
};
int main() {
    Year year{ 2024 }; // type int
    int val = year.value(); // matching type
}
```

Listing 12.30 The internal type “`int`” has become part of the class interface.

But what if you, as the developer of the `Year` class, later decide that `long` is a much better type for the encapsulated value? Then the developer who wrote `main()` has to adjust all their `Years` using code—even if that developer is you.

However, you and the user can take precautions together. Make the corresponding type *explicitly* part of the class interface and do not simply pass an essentially internal type to the outside. To do this, define a type alias locally but publicly in the class and use it in all relevant places.

```
// https://godbolt.org/z/EYjo8E8Gn
class Year {
public:
    using value_type = int; // introduce type alias
    value_type value_; // actually internally used type
public:
    explicit Year(value_type v) : value_{v} {}
    value_type value() { return value_; }
};
int main() {
    Year year{ 2024 }; // rely on compiler conversion here
    Year::value_type val = year.value(); // use ::
}
```

Listing 12.31 With “`using`”, you can introduce type aliases that make it easier to maintain interfaces than with the types themselves.

As you can see with `Year::value_type`, you need to use the *scope operator* `::` to access `value_type` from `Year`.

This way, you can always declare the variable that receives the return value of `value()` appropriately.

However, I would like to point out that this is really just an alias for the actual type used. This means you could still write `int val = year.value();` with internal knowledge of the class and not get an error from the compiler. If you have such specialized knowledge, you should refrain from using it. Use the publicly exposed type alias as often as possible. This makes your code more flexible. It also indicates that you chose this type for a specific reason—to match the class being used—and not just because you particularly like `int` or because it is 32-bit.

The classes of the standard library also offer you this mechanism. If you have forgotten the element type of your vector, there is a vector-local `value_type` type alias for that. Perhaps more useful is `size_type` for the type that `size()` and similar methods return.

```
// https://godbolt.org/z/zdMfoa7Yj
#include <vector>
#include <set>
#include <iostream>
using std::vector; using std::set; using std::cout;
using vector_t = vector<unsigned long long>; // Your own type alias
int main() {
    vector_t huge{ 12ULL, 10'000'000'000ULL, 9ULL, 0ULL,  };
    vector_t::size_type sz = huge.size();
    vector_t::value_type uiuiui = huge[1];
    for(vector_t::iterator it = huge.begin(); it != huge.end(); ++it)
        *it *= 2; // double
    /* sort via set */
    set<vector_t::value_type> sorted{huge.begin(), huge.end()};
    for(vector_t::value_type val : sorted)
        cout << val << " ";
    cout << "\n";
}
```

Listing 12.32 The standard library also contains many useful type aliases.

So, you are always on the safe side when using internal types of the standard library. By introducing the custom `vector_t` type alias, I want to illustrate what I meant by “forgetting” the element type. By introducing `vector_t` as a name for `vector<unsigned long long>`, `unsigned long long` is somewhat hidden. Now it is even easier to write `vector_t::value_type` instead of scrolling back to the beginning of the file to check what the exact element type of the vector was back then.

For example, the standard containers offer an `iterator` type alias, whose actual type you as a user cannot easily guess.

With `set<vector_t::value_type>`, I want to demonstrate that you can use type aliases for more than declaring new variables. Here, the element type of the `set` should, for example, correspond to the element type of the `vector`. And if that's what you mean, why not say it? `set<unsigned long long>` would not have caused a compilation error, but by using `vector_t::value_type`, I have precisely stated what I meant: that is, please use the same type.

One more word about the function of the example: I used the standard `set` container because it always keeps its elements sorted. Like `vector`, you can also initialize it with two iterators as arguments. The `set` then copies the entire range between the iterators into itself and sorts them efficiently. To observe the sorted order, you only need to look at its elements from `begin()` to `end()`—which the range-based `for` loop around `sorted` implicitly does.

12.12 Type Inference with “auto”

It would be even easier if you didn't have to look up exactly what `begin()` returns. The exact type doesn't really matter to you in this case. You define a variable `it` with the type and insert it into other functions. Someone who knows exactly what type `begin()` returns could actually help you and save you the trouble—and that someone is the compiler.

Suppose the compiler encounters something that looks like this:

```
vector<int> data{};
? it = data.begin();
```

Then instead of `?`, there really isn't much that can stand there. A look at the definition of `vector::begin()` shows that it can only be one of these two options:

- `vector<int>::iterator`
- `vector<int>::const_iterator`

It's even easier in the case of `size()`:

```
vector<int> data{};
? sz = data.size();
```

If you do not want implicit type conversion, only the following is applicable for the `?`:

```
vector<int>::size_type
```

Therefore, let the compiler determine the correct type. However, you do not use `?` for this, but the `auto` keyword.

```
// https://godbolt.org/z/Wh71cf7j1
#include <vector>
#include <set>
#include <iostream> // cout
using std::vector; using std::set; using std::cout;
using vector_t = vector<unsigned long long>; // Your own type alias
int main() {
    vector_t huge{ 12ULL, 10000000000ULL, 9ULL, 0ULL, };
    auto sz = huge.size();
    auto uiuiui = huge[1];
    for(auto it = huge.begin(); it != huge.end(); ++it)
        *it *= 2; // double
    /* sort via set */
    set sorted(huge.begin(), huge.end()); // set<vector_t::value_type>
    for(auto val : sorted)
        cout << val << " ";
    cout << "\n";
}
```

Listing 12.33 When initializing a variable, the compiler can determine the type.

As you can see, not many type specifications remain. You can use `auto` anywhere you initialize a variable with an expression. The compiler knows the type of the initialization expression and sets it as the type for the variable.

“auto” Permanently Sets the Type of the Variable

It is important to note that `auto` is equivalent to writing the type by hand. The type is really *set* and cannot be changed. It is not the case—as one might expect—that you create a variable with a mutable type. All uses of the variable have this type. There is no reassignment with a new type:

```
auto value = 12;           // value is now an int
value = string("Hello");  // ✗ Reassignment with a different type is not possible
```

However, an `auto` declaration of a variable might allow a bit too much freedom and tells the reader of the code less than an explicit type might. Therefore, since C++20, you can restrict `auto` with a concept that the variable must satisfy. The previous example could look like the following listing in that case.

```
// https://godbolt.org/z/qzWchWoek
#include <vector>
#include <set>
#include <iostream> // cout
#include <concepts> // integral
```

```

#include <iterator> // output_iterator, input_iterator
using namespace std;
using vector_t = vector<unsigned long long>; // your own type alias
int main() {
    vector_t huge{ 12ULL, 10000000000ULL, 9ULL, 0ULL, };
    unsigned_integral auto sz = huge.size();
    unsigned_integral auto uiuiui = huge[1];
    signed_integral auto meh = huge[1]; // ✗ concept not fulfilled
    input_or_output_iterator auto itx = huge.begin(); // concept without parameter
    for(output_iterator<unsigned long long> auto it=huge.begin();
        it!=huge.end(); ++it)
        *it *= 2; // double it
    /* sort using set */
    set sorted(huge.begin(), huge.end()); // set<vector_t::value_type>
    for(const unsigned_integral auto& val : sorted)
        cout << val << " ";
    cout << "\n";
}

```

Listing 12.34 Type deduction with “auto” can be further restricted using a concept.

The simplest case here is `unsigned_integral auto sz`, where you simply write the desired concept before `auto`. In the case of `signed_integral auto meh`, the concept does not match and an error occurs. Many iterator concepts have more than one parameter, so for `output_iterator<unsigned long long> auto it`, you need the element type as a parameter. Thus, it is now restricted as an iterator to which you can assign an unsigned long long with `*it`. I find it only moderately useful to write it this way because in this case you could have also dispensed with type deduction using `vector_t::iterator`. For `input_or_output_iterator auto itx`, a weaker concept is used that does not require an element type.

You can enrich `auto` with modifiers like `const` and `&` to adjust the final type. The reference can sometimes be especially crucial here. You then do not get a copy, but a reference that you can modify.

```

// https://godbolt.org/z/ro6fdsvT9
#include <vector>
#include <iostream> // cout
using std::vector; using std::cout;
int main() {
    vector data{ 12, 100, -1, 0, }; // vector<int>
    for(auto& val : data)
        val *= 2; // double it
    for(const auto val : data)

```

```
        cout << val << " ";  
    cout << "\n";  
}
```

Listing 12.35 When you enrich “auto” with &, you get a modifiable reference.

The first `auto` must be enriched with `&` so that you get a reference to the actual data that you want to modify with `val *= 2`. If you had only written `for(auto val : data)`—without `&`—then `val` would have been a copy each time, and `val *= 2;` would have had no effect on the content of `data`.

For the second `auto`, it is quite okay because you do not need to modify `val` for the output. And because you want to make sure that this does not happen accidentally, you can protect yourself from unintended changes by enriching with `const`. Thus, each element is copied to `val`, marked as constant, and output.

For `int`, this copy is fine. However, if the elements were immensely large and expensive to copy, you should have combined both enrichments. Assuming `Image` is a class that is expensive to copy, it would be advisable to use `const` and `&` together with `auto`:

```
vector<Image> data{ Image{"MonaLisa.png", "TheScream.png" };  
for(const auto& image : data)  
    show(image);
```

Returning to the original example, there are a few lines left where `auto` could not be used:

```
vector<_t huge{ 12ULL, 10000000000ULL, 9ULL, 0ULL,  };  
// ...  
set sorted( huge.begin(), huge.end() );
```

For `huge`, you cannot use `auto` because the compiler needs to know that you want to declare a `vector<unsigned long long>`. The compiler might guess the element type from the 12ULL list elements, but whether you want a `vector`, `set`, or something else entirely, the compiler cannot know.

Whenever you initialize with a list in curly brackets, you must explicitly specify the desired type. If you were to use `auto` here, the declared variable would get the internal `initializer_list<>` type, which is usually not what you want.

The same applies to `sorted`: here you do not have a single expression from which the compiler can deduce the type, but an explicit constructor call (with two iterators as arguments). In such a case, the compiler must already know from which class you want to call the constructor. This is not an initialization with a left side and a right side as `auto` would need to be able to do its job.

But the compiler helps you as much as it can. Since C++17, the compiler automatically determines the type parameter for `set` from the constructor arguments—without you

having to write `auto`. As you can see, you only need to write `set` rather than using a full type before C++17:

```
set<vector_t::value_type> sorted(huge.begin(), huge.end());
```

12.13 Custom Classes in Standard Containers

In this section, I refer to [Chapter 24](#). That means you will see the ubiquitous `vector`, the sorted `set`, and the mapping `map`. However, this generally applies to all containers. If you are not yet familiar with them, you should initially skip this section. But as soon as you want to make your own classes fit to be packed into containers, you should study this section.

You can put (almost) all your own classes into a `vector`. There are only a few conditions tied to the data type in the container: your class should be able to be created with the default constructor (without arguments), and you must be able to copy and assign it.

```
// https://godbolt.org/z/8ejPs6zr6
#include <vector>
struct Number {
    int value_ = 0;
    Number() {} // Default constructor
    explicit Number(int v) : value_{v} {}
};
int main() {
    std::vector<Number> numbers{}; // okay: Number meets the requirements
    numbers.push_back( Number{2} );
}
```

Listing 12.36 To pack a custom data type into a “vector”, it does not need to meet many requirements.

Because you do not define an assignment operator or copy constructor, the compiler tries to generate them for you—which succeeds because you do not have references or constants in the class (see [Chapter 16](#)). You need to define the default constructor yourself here as you have `Number(int)` as a custom constructor; the compiler would not generate `Number()` automatically.

A small note: if you are clever and restrict yourself regarding the operations you perform on the containers, you can even forgo some requirements for the element type. For example, you can forgo the default constructor if you do not use `numbers.resize()` or the like. Normally, however, it is a good idea to equip types in containers with a default constructor, copy constructor, assignment operator, and move operations.

Under these conditions, you can create the `vector<Number>`. The same applies to `array`, `deque`, `list`, and `forward_list`.

Collectively, these are called *sequence containers*. You determine the order of the elements by the explicit insertion position (see [Chapter 20](#)).

In contrast, the ordered associative containers are always in an order defined by the elements themselves. For `map` and `set` as well as their relatives `multi_map` and `multi_set`, the order depends on the order in which you inserted the elements. When you check, the elements are in a consistent position.

`map` and `set` achieve this by keeping the elements in the container in a sorted order at all times. The sorting is determined by the `operator<` free function. For example, because `operator<` is already defined for `int` and `string`, `set<int>` and `set<string>` work directly.

This does not apply to `Number` from [Listing 12.36](#). When attempting to create `set<Number>` and insert an element with `insert`, the compiler reports an error. You need to overload the `operator<` appropriately.

```
// https://godbolt.org/z/Yhzqo6dxT
#include <set>
struct Number {
    int value_ = 0;
    explicit Number(int v) : value_{v} {}
};
bool operator<(const Number& left, const Number& right) {
    return left.value_ < right.value_;
}
int main() {
    std::set<Number> numbers{}; // okay
    numbers.insert( Number{3} ); // operator< is needed here
}
```

Listing 12.37 For a “set” of a custom data type, you need to override “operator<”.

The same applies to `map`. The sorting is done based on the key elements—that is, the first of the two types. For this, you need to provide `operator<`. For the values, copy and assignment are sufficient, and preferably also the move operations.

```
// https://godbolt.org/z/TdEnhKxGK
#include <map>

struct Number {
    int value_ = 0;
    explicit Number(int v) : value_{v} {}
};

bool operator<(const Number& left, const Number& right) {
    return left.value_ < right.value_;
}
```

```
int main() {
    std::map<Number,int> numbers{};           // okay
    numbers.insert( std::make_pair(Number{4},100) ); // needs operator<
    numbers[Number{5}] = 200;                // here as well
}
```

In a map, the elements are stored as `std::pair`. To insert an element using `insert` similarly to set, you create a `std::pair<Number,int> pair`—preferably with `std::make_pair`, as shown.

You can also insert a new key-value pair using `operator[]`. This saves you from using `make_pair`. Internally, this is a bit more complex for the compiler, but in most cases, you can ignore that.

A small note on `operator<`: even though the standard containers only require you to define `operator<` for your data type, you should at least consider defining the other operators, `>`, `==`, `!=`, `<=`, and `>=`, if you define `operator<`. For such a group of operations, it is always practical to provide the complete set so that users do not have to struggle or look up what is available. At least if `operator<` is not the only operation, you should support the entire group.

If you have good reasons to implement only `operator<`, consider also implementing `operator==`, which is needed for *unordered associative containers*. If you have `operator<`, you can check “not `a < b` and not `b < a`” as the ordered associative containers do, but you might have a more efficient implementation for your data type.

12.13.1 Three-Way Comparison: The Spaceship Operator

With C++20, you can define *greater*, *smaller*, *equal*, *smaller-equal*, and so on for custom types without overloading a bunch of operators and potentially making mistakes. Simply define the `operator<=>`, and your data type will support the complete set of operators, especially `<` and `==`, to be used in ordered containers.

If you insert the following definition for a custom data type, then the compiler generates `operator<=>` such that the elements are compared in the order in which they are defined in the class:

```
auto operator<=>(const X& other) const = default;
```

If you desire different behavior, then implement the operator yourself.

A Custom-Defined `operator<=>` Also Requires a Custom-Defined `operator==`

If you define `operator<=>` yourself, you should also define `operator==` yourself. This is because if you do not let the compiler generate `operator<=>` with `= default`, then `operator==` will not be generated automatically. It will be missing, and you will not be able to use `==` for your type.

The reason is that the result of the `==` check can often be computed faster than the result of `<=>`, which covers the three `<`, `==`, and `>` operators. So the compiler assumes that you have a better implementation for `operator==` if you already know better than the compiler what is good for `operator<=>`. Also, read the box in [Chapter 24, Section 24.8.1](#).

With an `operator<=>` (and possibly `operator==`), the normal comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=` can then be derived by the compiler. This is done using the new mechanism of *rewritten candidates*.

The `operator<=>` does not need to be a friend. A special case applies to it, allowing its implementation to access private elements without `friend` and behave as if it were defined as a free function.

If you want to highlight that your data type does not support `operator<=>`, then you can declare the operator with `= deleted`:

```
struct X {
    Data data_;
    /* Comparisons would be too expensive and are not supported */
    auto operator<=>(const X& other) const = delete;
};
```

Even without the `auto operator<=>... = delete` line, comparisons would not be possible, but with the line, you make it clear to readers that you consciously do not support comparisons.

Chapter 13

Namespaces and Qualifiers

Chapter Telegram

- **namespace**
Introduces a new namespace, either named or anonymous.
- **namespace std**
In the `std` namespace, you will find all the identifiers of the standard library.
- **static**
Marks a variable, function, or method as file-local, shared, or persistent.
- **Singleton**
Popular design pattern in which only one instance of a type exists at most.
- **const**
Marks something as immutable at runtime.
- **constexpr**
Marks a constant or function as computable at compile time.
- **constexpr**
Lets the compiler guarantee that a function is computable at compile time.
- **Const-correctness**
By using `const` on types, you can get help from the compiler in uncovering some typical errors.
- **Type safety**
The correct use of types, including `const`, is enforced by the compiler with type safety.
- **volatile**
Marker for *volatile* variables that can change without the compiler's knowledge.

In this chapter, you will learn about two quite different things: `namespace` and `static`. However, due to the multiple meanings of many keywords in C++, there are overlaps. Therefore, I have decided to go from `namespace` to the overlap with `static` within one chapter.

13.1 The “std” Namespace

All things in the standard library start with `std::`; you have seen this often. `std::cout` and `std::vector` are all in the `std` namespace. And in [Chapter 4, Section 4.2](#), I explained how you can save the `std::` prefix for a single identifier with `using std::cout`; and for all identifiers with `using namespace std`;

But how does something get into a namespace in the first place? Simply enclose everything that should go into the namespace—let’s say `plant`—within `namespace plant {...}`. This opens a *scope*, which behaves similarly to a class regarding identifiers: Within the range, you can omit `plant` for access to other elements; from outside, you need the `plant::` prefix, as you can see in [Listing 13.1](#).

Because this is about namespaces and `using`, I have refrained from using `using std::string`, `using std::ostream`, and the like globally at this point. I do this in this book, as already explained, only for space reasons. In a real project, you should use each `using` at the global level very sparingly (up to complete abstinence).

```
// https://godbolt.org/z/qo6W3bh5d
#include <string>
#include <iostream>                                // ostream, cout
namespace plant {
    class Tree {
        std::string name_;
    public:
        explicit Tree(const std::string_view name) : name_{name} {}
        void print(std::ostream& os) const { os << name_; }
    };
    std::ostream& operator<<(std::ostream& os, const Tree& arg)
    { arg.print(os); return os; }
    using ConiferTree = Tree;                        // for future extensions ...
    using BroadleafTree = Tree;                     // ... provide for
    namespace exampleNames {                        // embedded namespace
        std::string oakName = "Oak";
        std::string beechName = "Beech";
        std::string firName = "Fir";
    } // end namespace exampleNames
} // end namespace plant

int main() { // main must not be in a namespace
    using namespace plant::exampleNames; // make all example names available
    plant::ConiferTree fir{ firName };
    plant::BroadleafTree oak{ oakName };
    fir.print(std::cout); std::cout << "\n";
```

```

    using plant::operator<<;           // without it 'cout << oak' won't work
    std::cout << oak << "\n";
}

```

Listing 13.1 You define a namespace with “namespace”.

I have packed all sorts of things into the `plant` namespace for demonstration purposes:

- The `plant::Tree` custom data type
- A `operator<<` free function
- Two type aliases, `ConiferousTree` and `BroadleafTree`
- Another namespace

The two type aliases are meant to suggest that you might plan to separate the classes `ConiferousTree` and `BroadleafTree` from `Tree` in the future and write custom classes for them, but for now they are still identical.

You can, if it seems sensible to you, nest namespaces almost arbitrarily and group your project this way. I have packed another element into the `exampleNames` embedded namespace: variables such as `oakName` and the like.

You can also place constants and templates in a namespace. Only macros are not affected by namespaces; see [Chapter 21](#).

`main` must be outside of any namespace; otherwise, the compiler will not find the function as an entry point. In this `main`, I refer to `plant::ConiferTree` and `plant::BroadleafTree` with the namespace qualifier. Alternatively, the following would have worked:

```

using plant::ConiferTree; using plant::BroadleafTree;
ConiferTree pine{ pineName };
BroadleafTree oak{ oakName };

```

Or simply:

```

using namespace plant;
ConiferTree pine{ pineName };
BroadleafTree oak{ oakName };

```

I also chose this approach for using namespace `plant::exampleName`. As a reminder: you should only include an entire namespace with `using namespace` locally—for example, within a function—but never globally in a `*.cpp` file—and certainly not in a header file.

Because `fir` is a local variable in `main`, you can call `print` as a method:

```

fir.print(std::cout); std::cout << "\n";

```

It's a bit trickier with `operator<<`. If you simply write `std::cout << tanne`, it will cause an error because the compiler looks for a suitable free function in all available namespaces (for operators, `std` is also searched). So it looks for the following:

- `::operator<<(std::ostream&, const Tree&);`—globally, the level where `main` is
- `std::operator<<(std::ostream&, const Tree&);`—where `std` is searched
- `plant::exampleNames::operator<<(std::ostream&, const Tree&);`—because using namespace `plant::exampleNames` was previously declared

The overload for `Tree` is in namespace `plant`:

```
plant::operator<<(std::ostream&, const Tree&);
```

To use an operator defined within a namespace, you must first make it available so that it is included in the search. There are two options here:

- `using plant::operator<<;`—to only fetch the operator
- `using namespace plant;`—to fetch everything from the namespace

Namespaces are also useful for separating things that might otherwise conflict. Operators are good candidates here. You might have two `operator<<` variants, as in the following listing.

```
namespace plant {
    // ... as before ...
    std::ostream& operator<<(std::ostream&, const Tree&) {...};
    namespace debug {
        std::ostream& operator<<(std::ostream&, const Tree&) {...};
    }
}
plant::Tree tree{"MyTree"};
void run() {
    using namespace plant;
    cout << tree << "\n";
}
void diagnostic() {
    using namespace plant::debug;
    cout << tree << "\n";
}
int main() {
    run();
    diagnostic();
}
```

Listing 13.2 In separate namespaces, you can define the same operators.

Depending on the need, you include the desired `operator<<` using a `using` directive. Unlike functions or types, you cannot specify the scope with operators; `plant::Tree tree{"x"};` works, but `cout plant::<< tree;` does not work. Instead, you could write

`plant::operator<<(cout, tree)`, but that makes operators pointless, as you might as well define a function or method.

So if you are tempted to always define operators in the global namespace, you don't need to do that if you know that `using plant::operator<<` also helps.

Use Namespaces

If you want to structure your project cleanly, use namespaces. I recommend using one main namespace per custom library. Whether you want to subdivide this namespace further depends on the requirements.

By the way, you can extend a namespace later without any problem and split it into two or more files. Simply put everything you want in each file into `namespace plant {...}`, and the namespace will fill up from file to file.

13.2 Anonymous Namespace

You can also define a namespace without its own identifier. This then has a special meaning. For the purposes of the next listing, assume that you have several files here.

```
// https://godbolt.org/z/e3d55zvT8
// modul.hpp
#include <string>
#include <iostream>
namespace plant {
    class Tree {
        std::string name_;
    public:
        explicit Tree(const std::string_view name);
        void print(std::ostream& os) const;
    };
    std::ostream& operator<<(std::ostream& os, const Tree& arg);
}

// modul.cpp
#include "module.hpp"
namespace { // anonymous namespace
    std::string PREFIX = "TREE:";
    void printInfo(std::ostream& os) {
        os << "Author: Torsten T. Will\n";
    }
}
```

```
bool debug = false; // global, no namespace
namespace plant {
    Tree::Tree(const std::string_view name)
        : name_{name} {}
    void Tree::print(std::ostream& os) const {
        os << PREFIX << name_;
    }
    std::ostream& operator<<(std::ostream& os, const Tree& arg) {
        if(debug) printInfo(os);
        arg.print(os); return os;
    }
}

// main.cpp
#include "module.hpp"
int main() {
    plant::Tree x{"x"};
    x.print(std::cout); std::cout << "\n";
}
```

Listing 13.3 An anonymous namespace makes definitions local to the current file.

You see three files here:

- `module.hpp`—interface to class `Tree` with the method declarations
- `module.cpp`—implementation of the `Tree` class, with some helpers in an anonymous namespace *and* a global `debug` variable
- `main.cpp`—uses the interface

So, you have two modules and a header. If you were to list all the identifiers defined in all modules, you would find that there are no overlaps. There must not be any. All defined identifiers of all modules must be distinct from each other.

For example, you could not define a `int debug = 55;` global variable in `main.cpp`. The compiler would detect during linking that the identifier `debug` exists multiple times and would produce an error. This is because in `modul.cpp`, a `debug` is already defined; it is not in any namespace.

The set of identifiers can become very confusing in large projects, especially when you also use third-party libraries. You get a large number of identifiers delivered with them, which your identifiers must also not overlap with.

The first precaution for this is to use namespaces. The identifier of the namespace is part of the entire identifier name. This way, `plant::Tree` and `algo::Tree` would not interfere with each other.

The second measure is to use an *anonymous namespace* for what is not needed in other modules. `main.cpp` is not concerned with `PREFIX`. And the `printInfo` function is only used within `modul.cpp`. By encapsulating these two identifiers within a namespace `{...}`, they are only visible within the current module. They no longer interfere with another module.

13.3 “static” Makes Local

For exactly this purpose, you can also use the keyword `static`. Instead of opening an anonymous namespace, you write `static` before the variables and functions. You achieve the same thing: the name of the variable or function is only visible within this module. Other modules can use the same name without the compiler complaining.

It would look like this for `modul.cpp`:

```
// https://godbolt.org/z/zfYzPjqrW
// modul.cpp
#include "modul.hpp"
static std::string PREFIX = "TREE:";
static void printInfo(std::ostream& os) {
    os << "Author: Torsten T. Will\n";
}
bool debug = false;
// rest as before
```

The `debug` variable is not yet `static`. As a result, no other module can define a global function or global variable `debug`, but that is intentional here. Because although `debug` is not in the `modul.hpp` header, someone with enough insider knowledge (the author?) can manipulate the variable. Consider this an excursus as it should not be part of usual programming practice to hide and change variables in this way. In some other module of the project, someone can manipulate the variable with

```
extern bool debug;
```

and declare and then manipulate it with `debug = true;`. This can be both good and bad. For a debugging switch, it might just be permissible. Otherwise, put identifiers that should not be exported into an anonymous namespace (or declare them `static`).

An anonymous namespace has the great advantage over `static` that it also protects its own types from spilling over into other modules. You cannot protect `struct`, `class`, or using `X = Y` from export with a `static`. Within a namespace `{...}`, however, these things are safe.

13.4 “static” Likes to Share

The fact that `static` is also used for a completely different purpose takes some getting used to. You can use it to mark a data field or a method of a class to share it among all instances of the class. Or to put differently:

- **static for a data field**

While there is always one instance of a normal data field per instance, all instances share a common data field marked as `static`. Therefore, there is exactly one instance of this data field per class and not as many as there are instances.

- **static for a method**

A static method lacks the `this` pointer. Within the method, there is no connection to a specific instance of the class. Therefore, you do not call the method via an instance with `.` or `->`, but rather via the class name with `::`.

Both variants are similar in that you can access both even if you have not created a single instance yet.

Typical applications are as follows:

- A static data field can count the number of active instances of the class.
- A static method can serve as a *factory* for controlled creation of instances (popular design pattern).

In the following listing, both application possibilities are in use.

```
// https://godbolt.org/z/n4e8xdEGj
#include <iostream> // cout
#include <string>
using std::string;
class Tree {
    static size_t countConstructed_;
    static size_t countDestructed_;
    string kind_;
    Tree(string kind) : kind_{kind}      // private constructor
    { ++countConstructed_; }
public:
    Tree(const Tree& o) : kind_{o.kind_}
    { ++countConstructed_; }
    string getKind() const { return kind_; }
    ~Tree() { ++countDestructed_; }
    static Tree create(string kind) { return Tree{kind}; }
    static void stats(std::ostream& os) {
        os << "Constructed:+" << countConstructed_
            << " Destructed:-" << countDestructed_ << "\n";
    }
};
```

```

size_t Tree::countConstructed_ = 0;
size_t Tree::countDestructed_ = 0;

int main() {
    Tree birch = Tree::create("Birch");
    for(auto kind : {"Ash", "Yew", "Oak"}) {
        Tree temp = Tree::create(kind);
        std::cout << temp.getKind() << "\n";
    }
    Tree::stats(std::cout);
}

```

Listing 13.4 All instances share their “static” data fields and methods.

First, I declared two counters `countConstructed_` and `countDestructed_`. Unlike normal data fields, you unfortunately cannot initialize static data fields within the class. Therefore, you see a repetition of the static data fields with the initialization values outside the class—here, 0 for each.

Note that if the class is defined in a header that may be included by multiple *.cpp files, you must place the initialization definitions in a *.cpp file. Otherwise, the initialization would be performed multiple times in the entire program, or the linker would complain about duplicate definitions.

The static `create` method is in the public section of the class. This should now be used to create new `Tree` instances. To prevent anyone from calling it from outside, the constructor has been moved to the private section of the class.

Within the constructor, I increment the `countConstructed_` counter. To increment the `countDestructed_` counter, I also need the destructor. This should almost always be in the public section of the class.

`Tree` still needs the copy constructor because `create` returns the newly created tree as a copy. Copying also creates a new object that should be counted with `countConstructed_`, so the = default copy constructor is not suitable. As the class currently stands, you could also do without the copy constructor and instead define the move constructor, but that's just a detail.

You see in `main` that you can call `Tree::create` without a `Tree` instance already being created. When you are inside the `create` method, there is no instance whose data fields you can access. You can only access other static data fields and static methods—and constructors.

There is a distinction between calling a static `Tree::create()` method and a normal `tree.getKind()` method: the use of the class name instead of the instance variable, as well as the scope operator `::` instead of the access point `.` or the access arrow `->`, if it is a pointer to an instance.

Similarly, I call `Tree::stats()` at the end. Again, only static data fields are accessed. And the output

```
Constructed:+4 Destructed:-3
```

proves that each instance accessed the same counter variables in its constructor or destructor. The count is correct because all `Tree temp` objects in the loop were also destroyed again. Only the local `birch` variable still exists at the time of the `stats()` call.

Copy Constructor and Move Constructor

How to make a class that copies itself is detailed in [Chapter 16](#). At this point, you should know that classes can be copied. Sometimes the compiler can set the rules for this, and sometimes you want or need to do it yourself. To do this, declare this constructor:

```
Keyboard(const Keyboard&);
```

In the definition, you then specify how the copying should be done. If you explicitly want to prevent a copy from being possible, write the declaration like this:

```
Keyboard(const Keyboard&) = delete;
```

The same applies to moving. The declaration that you use to declare and simultaneously prevent the move is this:

```
Keyboard(Keyboard&&) = delete;
```

However, this is implicit if you have specified the copy constructor. For more on this, see [Chapter 16](#).

13.5 Remote Initialization or “static inline” Data Fields

Assume you have defined a `Tree` class in the `tree.hpp` header and used the header multiple times in your program. Then you must ensure that the initialization of the static data fields is done in exactly one `*.cpp` file and not in the `*.hpp` file as they would otherwise be defined multiple times.

Therefore, starting from C++17, in addition to `static`, write the `inline` keyword and set the field directly—that is, the declaration and definition together. Then the compiler ensures unique initialization:

```
class Tree {
    static inline size_t countConstructed_ = 0;
    static inline size_t countDestructed_ = 0;
    // ...
}
```

13.6 Guaranteed to Be Initialized at Compile Time with “constinit”

Starting from C++20, you can mark global or static variables with `constinit`. This ensures that the compiler guarantees that the variable can be initialized at compile time. The chances are then high that no initialization work needs to be done at run-time.

Even though the sequence of letters *const* appears in this keyword, the variable is not constant. One could say that “constinit is `constexpr` minus `const`”:

```
// https://godbolt.org/z/b9arcEMM1
constinit auto SZ = 10*1000-1;           // global variable
size_t autoincrement() {
    static constinit size_t i = 0;       // local static variable
    return i++;
}
class BraitenbergVehicle {
    inline static constinit size_t count_ = 0; // class variable
public:
    size_t id_;
    BraitenbergVehicle() : id_{++count_} {}
    ~BraitenbergVehicle() { --count_; }
};
```

The `SZ`, `i`, and `count_` variables are initialized at compile time. The `i` and `count_` variables are also modified during the program's execution.

In practice, `constinit` solves the problem in which sometimes the order of initialization of static variables is not clear, and you might not realize that you are accessing a variable that has not been initialized yet. With `constinit`, it is guaranteed that the expression can be evaluated at compile time and is not affected by such problems.

You can use `constinit` together with `extern` and `thread_local`.

13.7 “static” Makes Permanent

Similar, but not the same, is the use of `static` with local variables—that is, within a function. When you add `static` to the declaration of a local variable, this variable persists when the program flow leaves the scope and is reused when it is entered again.

```
// https://godbolt.org/z/h5qond6db
#include <iostream>           // cout
class Keyboard {
    Keyboard(const Keyboard&) = delete; // no copy
    const size_t nr_;             // current number
};
```

```
public:
    static inline size_t count_ = 0;    // counts created instances
    explicit Keyboard() : nr_{count_++} {
        std::cout << "  Keyboard().nr:"<<nr_<<"\n";
    }
};

Keyboard& getKeyboard() {
    std::cout << "  getKeyboard()\n";
    static Keyboard keyboard{};        // static local variable
    return keyboard;
}

void func() {
    std::cout << "kbFunc...\n";
    Keyboard& kbFunc = getKeyboard();
}

int main() {
    std::cout << "kbA...\n";
    Keyboard& kbA = getKeyboard();
    func();
    std::cout << "kbB...\n";
    Keyboard& kbB = getKeyboard();
    std::cout << "count:" << Keyboard::count_ << "\n";
}
```

Listing 13.5 A local static variable is initialized once and reused thereafter.

Here you see the implementation of the so-called Meyer's Singleton pattern. A *singleton* is a way to always get the same instance of a class with each use. The uses can also come from different parts of the program, as seen here from `func()` and from `main()`; because both use the `getKeyboard()` free function, they also receive the same `Keyboard&` reference and thus the same object.

Singletons are particularly popular in programs with multiple threads. For example, imagine that your program has multiple *threads* (simultaneously running program strands), and each wants to read from the keyboard—and for that, it needs a `Keyboard` class. The `Keyboard` class establishes the connection to the one keyboard actually connected to the computer. It would be helpful if it were always the same keyboard. This is what the `getKeyboard()` function is supposed to provide in the example.

```
Keyboard& getKeyboard() {
    cout << "  getKeyboard()\n";
    static Keyboard keyboard{}; // static local variable
    return keyboard;
}
```

Listing 13.6 The Meyers Singleton.

In `getKeyboard()`, you see the static local `keyboard` variable. Unlike *global* static variables, this one is not initialized beforehand at program start (or even earlier), but exactly at the first passage of the definition.

And unlike nonstatic local variables, the static variable is not cleared when leaving the block. The block here is the `getKeyboard()` function. The `keyboard` variable still exists upon reentry and is not reinitialized but reused.

There are many variants of the singleton design pattern and even more different implementations with various advantages and disadvantages. Some programmers even go so far as to say that the singleton itself is bad practice (an “antipattern”). This discussion is out of scope for this book, but it is true that you should not overuse the singleton design pattern.

Specifically, regarding the implementation of the singleton using a static local variable: the Meyer’s Singleton pattern has the advantage of being simple to implement and the disadvantage of not providing complete protection against multiple `Keyboard` instances. You could still create your own `Keyboard{}` using the public constructor. You could address this with a slightly modified interface if you find it to be a significant disadvantage.

The use of static local variables is also safe with multiple threads: It is guaranteed that only one thread initializes the variable.

13.8 “inline namespace”

The last way to define a namespace is together with `inline`. You create an *inline namespace* as in the following listing.

```
// https://godbolt.org/z/115M1csEE
#include <iostream>
namespace mylib {
    namespace v1 {
        int version() { return 1; }
    }
    inline namespace v2 { // current version
        int version() { return 2; }
    }
}

int main() {
    std::cout << "Version " << mylib::version() << "\n";    // Output: 2
}
```

```
std::cout << "Version " << mylib::v1::version() << "\n"; // Output: 1
std::cout << "Version " << mylib::v2::version() << "\n"; // Output: 2
}
```

Listing 13.7 The identifiers of an inline namespace also go into its surrounding namespace.

With the `inline before namespace`, you open a normal namespace. In addition, all identifiers are also included in the surrounding namespace. In this way, you can version your own library, for example: you pack the latest behavior into an `inline namespace`, making it the default behavior. You can see this in [Listing 13.7](#) with `mylib::version()`.

If users need old behavior, they can explicitly access an inner namespace, here with `mylib::v1::version()`. The new version is also available in an inner namespace, `mylib::v2::version()`, if users explicitly request it.

The standard library provides further examples of `inline namespace`. You can find a list in [Chapter 28, Section 28.4.2](#). For example, `std::literals` and `std::literals::string_literals` are both `inline`. To include operator `"s"` from `string_literals`, you can use (among other things) one of the following `using ... options`:

```
{ // direct addressing:
    using namespace std::string_literals;
    auto str = "abc"s;
}
{ // use inline namespace string_literals:
    using namespace std::literals;
    auto str = "abc"s;
}
{ // use both inline namespaces:
    using std::operator"s";
    auto str = "abc"s;
}
```

13.9 Interim Recap

Because there can be so much confusion with `namespace`, `using`, and `static`, I will recapitulate the various usage scenarios here. For completeness, I will also include the type alias with `using`.

Language Element	Description
<code>namespace xyz {...}</code>	New namespace <code>xyz</code>
<code>namespace {...}</code>	Anonymous namespace; all identifiers file-local

Table 13.1 Variants of “`namespace`”, “`static`”, and “`using`”.

Language Element	Description
<code>inline namespace xyz {...}</code>	Embedded namespace
<code>static global variable</code>	File-local variable
<code>static free function</code>	File-local function
<code>static data field</code>	Shared between all instances of the class
<code>static method</code>	Method that you call without a specific instance
<code>static local variable</code>	A variable that persists beyond the scope
<code>using namespace xyz</code>	Import all identifiers from namespace <i>xyz</i>
<code>using xyz::abc</code>	Import identifier <i>abc</i> from namespace <i>xyz</i>
<code>using new = old;</code>	Introduce type alias <i>new</i> for type <i>old</i>

Table 13.1 Variants of “namespace”, “static”, and “using”. (Cont.)

13.10 “const”

The `const` keyword should not be underestimated in C++. When used correctly, it can be very useful to programmers. With it, the compiler can help avoid careless mistakes and discover design problems.

In C++, you can use `const` in various places: whether passed as a parameter, as a local variable, or as a data field of a class. Here is a (not exhaustive) explanation of what is declared as “immutable” by a `const`:

- `static const int MAX = 100;`
Together with `static`, a constant is defined.
- `const string& getName();`
The return value must not be changed.
- `void print(const string& msg);`
The parameter is not changed within the function.
- `void Widget::drawYourself() const;`
The method does not change its object.

The last item has a special status, so to speak. It promises that the `drawYourself` method of the `Widget` class does not change the state of the instance it is called on within its function body. Technically speaking, this defines the `this` pointer within the method as `this const * const`, whereas a method without this trailing `const` can change the state: `this * const`.

Yes, there is indeed still a `const` in there. It is crucial where you write the `const` for the type.

The following list illustrates the subtle but important differences in usage together with pointers `*` and references `&`:

- **`int const val`**
The `int` value `val` is immutable.
- **`MyClass const &obj`**
All data fields and methods in `MyClass`, of which `obj` is an object, are immutable.
- **`int const *p_int`**
The `int` value pointed to by the pointer is immutable.
- **`int *const p_int`**
The pointer `p_int` is immutable; the `int` value is mutable.
- **`char const * const cstr`**
Both the content and the pointer are immutable.

As a mnemonic, one could say that `const` always marks the entity as immutable *behind* which it stands. In the much more common notation, where `const` is mentioned first, you must mentally “move” the leading `const` one position to the right. The following notations are equivalent:

- **`const int val` and `int const val`**
- **`const MyClass &obj` and `MyClass const &obj`**
- **`const int *p_int` and `int const *p_int`**
- **`const char * const cstr` and `char const * const cstr`**

13.10.1 Const Parameters

You have already learned in [Chapter 4, Section 4.2](#) that you can mark a parameter with `const`, which often occurs together with call-by-reference passing of parameters with `&`:

```
void print(const vector<int> &primes) {  
    for(auto prime : primes) {  
        cout << prime << " ";  
    }  
}
```

The parameter `primes` cannot be modified now. Any modification such as `primes[7] = 4` or `primes.push_back(12)` will result in a compiler error. With classes and custom types, `const MyClass& arg` is often chosen as the passing method because the parameter does not need to be copied.

In the case of `MyClass arg`, it would be copied. You could safely modify `arg` within the function—because it is a copy, the changes do not affect the original object outside the caller—but copying a large object is costly.

If avoiding a copy is worthwhile, it is best to work with a `const&`. The `&` avoids the copy, and the `const` prevents accidental modifications to the external object.

What applies to classes is no different for built-in types. However, their copy costs are negligible, which is why you rarely see `const int&` or `const double&`. The same rules apply as for classes. The compiler prevents changes to the values:

```
void nonsensicalParameter(const int& arga, const double& argb) {
    arga = 7;    // ✗ Assignment to constant
    argb = 3.14; // ✗ Assignment to constant
}
```

As mentioned, you can do this with built-in types, but it is rather rare. Built-in types are almost always passed by value:

```
void valueParameter(int arga, double argb);
```

Then you can change them in the function, which has no effect outside. Or you pass them by reference if the change should affect the outside:

```
void input(int &arga, double &argb);
```

But it is better to choose return values here, for example:

```
std::pair<int,double> input();
```

In [Chapter 28, Section 28.1](#) on pair and tuple, you will see in more detail how this works with tuple.

Finally, you can also choose value passing (which, as you know, has its advantages) *and* make this value constant:

```
void constantValues(const int arga, const double argb);
```

The effect is that you get a copy when calling and at the same time ensure that you do not accidentally try `arga = 5`.

13.10.2 Const Methods

The compiler directly recognizes if you try to change an `int`. However, it becomes more difficult when you have an object equipped with methods:

```
void print(const vector<int> &primes) {
    if(primes.size() > 100)    // that's fine
        return;
    for(auto prime : primes) { // that's fine too
        cout << prime << " ";
    }
}
```

```
    }  
    primes.push_back(4);    // ✗ that's not fine  
}
```

How does the compiler know that `primes.size()` can be called on the `const&` but that, for example, `primes.push_back(4)` should result in an error message? This happens because the `size()` method of `vector` is marked with a trailing `const`—essentially:

```
size_t vector<int>::size() const;
```

This tells the compiler that this method call is allowed on a `const` object.

For a custom data type `Widget`, it might look like this:

```
// https://godbolt.org/z/WTWxYa3rd  
class Widget {  
    unsigned x = 0, y = 0, w = 0, h = 0; // for example  
public:  
    unsigned getLeft() const;  
    unsigned getTop() const;  
    unsigned getRight() const;  
    unsigned getBottom() const;  
    void setWidth(unsigned w);  
    void setHeight(unsigned h);  
};
```

If you now pass `Widget` as a `const` object (reference) into a function, you can call the `get` methods:

```
void show(const Widget& widget) {  
    drawBox(  
        widget.getLeft(), widget.getTop(),  
        widget.getRight(), widget.getBottom()  
    );  
}
```

The attempt to call `widget.setWidth(100)`; in `show` will be met with a compiler error as the method is not marked as `const`.

At the same time, the trailing `const` has another effect: you cannot change data fields within the implementation of such a method. This also includes calling a non-`const` method:

```
unsigned Widget::getLeft() const {  
    x = 77;    // ✗ data fields cannot be changed  
    setWidth(88);    // ✗ do not call non-const methods  
}
```

Taken together, these three effects are a very important tool for the compiler to detect careless errors and design problems. You should be careful when marking methods as `const`. You will find that as a result, you will also need to carefully either make parameters `const` or pass them by value. Accidental `&s`, which open the door to careless errors, are then no longer possible.

Once again for practice: Technically speaking, the implicit `this` parameter is transformed by the trailing `const` from `Widget* const this` to `Widget const* const this`. Can you already read the `const`s in the different positions? The `const` refers to `Widget`, which precedes it. Thus, everything in `Widget` is off-limits for changes within the method.

13.10.3 “const” Variables

You can also apply `const` to variables. This applies to both local variables in a function or globally as well as to data fields in a class. A `const` variable can only be initialized, but you cannot assign it a new value or change its value (through a non-`const` method call):

```
class Widget {
    const int id_;
public:
    explicit Widget(int id) : id_(id) {}
    Widget() : id_(0) {}
    void reset();
};
const Widget widget_a = Widget{12};
```

After these lines of code, you cannot write `widget_a = Widget{24};`. In the implementation of `reset()`, the compiler also forbids

```
void Widget::reset() {
    id_ = 36;
}
```

because `id_` is `const`—just as if you had passed it as a parameter to the function.

Use `const` variables when it seems sensible to you, with discretion. For variables that are valid in a long section of code, where it is easy to lose track of which variables exist at all, a `const` can be an effective protection against accidental modification.

```
// https://godbolt.org/z/bxPaz9aW5
#include <vector>
namespace { // anonymous namespace for constants
    const unsigned DATA_SIZE = 100; /* number of elements in data */
    const double LIMIT = 999.999; /* max value during initialization */
};
```

```
std::vector<int> createData() {
    std::vector<int> result(DATA_SIZE);
    double currVal = 1.0;
    for(auto &elem : result) {
        elem = currVal;
        currVal *= 2;           // next value is larger
        if(currVal > LIMIT) {
            currVal = LIMIT;    // no value should be larger
        }
    }
    return result;
}
```

Listing 13.8 Local constants of a file fit well into an anonymous namespace.

This is especially true for global variables that, for example, define certain program parameters. It is best to put them in an *anonymous namespace*. The effect of this is that the identifier cannot be seen by other modules; in large projects, this is important to prevent global variables and constants from conflicting.

This allows you to easily tweak certain sizes of the program while also documenting the size well.

13.10.4 Const Returns

Normally, functions return a *value*, not a reference. This means that a copy of what you had in the function is returned. At least, you can think of it conceptually this way. In reality, the compiler can often avoid the copy (see [Chapter 12, Section 12.11](#)):

```
// https://godbolt.org/z/fM4nsx9GP
struct Widget {
    int num_ = 0;
    void setNumber(int x) {    // a non-const method
        num_=x;
    }
};

Widget createWidget() {       // Return by value
    Widget result{};          // Create
    return result;
}

int main() {
    Widget w = createWidget(); // Return by value creates a copy
    w.setNumber(100);          // changing is naturally okay, w is non-const
}
```

Because a copy of `result` exists outside of `createWidget()` in `main()` with `Widget w = createWidget();`, it is no longer the function's concern whether its content is modified. Therefore, it makes little sense to also mark the return type with `const`.

```
// https://godbolt.org/z/98x3YsnKc
const Widget createWidget() { // return as const value
    Widget result{};
    return result;
}
int main() {
    Widget w = createWidget(); // copied into new non-const w
    w.setNumber(100);          // w is non-const, changing is okay
}
```

Listing 13.9 Although the return type is marked with “const” here, it has no effect because it is always copied.

Although the return type is `const Widget`, the `const` can never have an effect because the return is always copied as a value—and the caller has control over where it is copied. Here it is copied again into `Widget w` (non-const), and thus the caller can do whatever they want with *their* copy.

It is quite different with *references* (and pointers): a reference means “no copy.”

Pay Attention to the Lifetime of the Referenced Object

I can't say it often enough: if you return a reference, you must ensure that the object the reference points to exists for as long as you use it externally.

```
// https://godbolt.org/z/3EG1ovcev
#include <string>
#include <string_view>
using std::string; using std::string_view;

class Widget {
    string name_ = "";
public:
    void setName(string_view newName) {
        name_ = newName;
    }
    const string& getName() const { // const& return
        return name_;
    }
};
```

```
int main() {
    Widget w{};
    w.setName("Title");
    string name1 = w.getName();           // new string, thus a copy
    name1.clear();                        // you are allowed to modify the copy again
    const string& name2 = w.getName();    // const reference to internal string name_
    /* name2.clear(); */                  // name2 is const, so it doesn't work
    string& name3 = w.getName();          // ⚡ Function returns const&, not &.
    auto name4 = w.getName();             // identical to name1
    const auto& name5 = w.getName();      // identical to name2
}
```

Listing 13.10 Constant references in returns.

In the preceding listing, I call a function five times that returns a `const string&`. The caller should store the result in variables of different types:

- **string at `name1 = w.getName();`**
Here `const string&` is converted to a `string`, which means the compiler should copy the return value. And as previously explained with the return of values, the caller has control over the copy.
- **const string& at `name2 = w.getName();`**
This is exactly the type that is also returned, so no conversion is necessary. A `const&` points to exactly the same `name_` that exists in `w`. If `w.name_` changes due to any influences, it would also be visible in `name2`. Only `name2` itself cannot be changed from the outside because it is a `const` reference.
- **string& at `string& name3 = w.getName();`**
Are you thinking “But I want to change `w.name_`” and therefore wanting to try to store the return value in a `string&`? The compiler does not allow this; a conversion from a `const&` to a `&` is not permitted. This is not like `name1`, where a `const& string` was converted into a `string` by copying it.
- **string with `auto name4 = w.getName();`**
The compiler sets `auto` to the type `string`. Thus, the type of the variable is `string`.
- **const string& with `const auto& name5 = w.getName();`**
Again, the compiler substitutes `string` for `auto`. The overall type is then `const string&`.

When a class returns a reference, it is quite common to also mark it with `const`. You often do not want external changes to values that reside within the class. Therefore, you first choose to return by value, and if that is inappropriate, by reference. Whether it is `const` or not depends on the intended use.

For reference returns, both are conceivable. Sometimes you want the value to be changeable from outside, sometimes not.

```

// https://godbolt.org/z/9MTj3rP5d
#include <string>
#include <iostream>
using std::string; using std::cout;

class Widget {
    string name_{};
public:
    const string& readName() const;           // const&-return, const-method
    string& getName();                       // &-return
};

const string& Widget::readName() const { return name_; }
string& Widget::getName() { return name_; }

int main() {
    Widget w{};
    const string& readonly = w.readName(); // const&, immutable
    cout << "Name: " << readonly << "\n"; // still "" empty.
    string& readwrite = w.getName();       // &, mutable
    readwrite.append("attached");           // also changes name_ and readonly
    cout << "Name via readwrite: " << readwrite << "\n"; // "attached"
    cout << "Name via readonly: " << readonly << "\n"; // also "attached"
}

```

Listing 13.11 References can be returned as constant and nonconstant.

Because both `readonly` and `readwrite` are references to `w.name_`, changing `readwrite` affects `w.name_` and also `readonly`.

Note that the `readName()` method, which has a `const&` return type, is also marked with a trailing `const`: because you can be sure that the return value will not be modified by anyone, calling this method does not change the state of the object—exactly what the trailing `const` signifies.

It's different with `getName()` with its `&` return type (non-`const`): you cannot mark this method with a trailing `const`. Because a part of the internal state (`name_`) is exposed through reference return, the internal state is no longer under the control of the method. Such a method can therefore never be `const`.

I have intentionally named the two methods differently in this example. However, you can also name them the same—that is, *overload* them. The compiler will then choose the appropriate variant:


```
class Widget {
    string name_{};

public:
    const string& getName() const;
    string& getName();
};
```

Depending on whether you are allowed to modify the return value at all, the appropriate overload will be chosen. If the `Widget` variable itself is `const`, the compiler will choose the first variant. Or in other words, the `Widget` variable on the right side of the assignment is relevant, not the `string` on the left.

Remember: `*this` is a hidden parameter in the method call—and the trailing `const` is its parameter type after `Widget const*` `const`. This way, the compiler can also call one method or the other depending on the `const` property of the actual `Widget`:

```
Widget wid{};
cout << wid.getName(); // chooses non-const, because wid is not const
const Widget vot{};
cout << vot.getName(); // chooses const, because vot is const
```

For starters, I recommend avoiding such overloads and making the distinction clear in the function name. You will find some overloads of this kind in the standard library, such as `vector::front()`.

13.10.5 “const” Together with “static”

You will often see that a `const` variable is also marked with `static`. Let me show you three variants:

```
// https://godbolt.org/z/4dnTs4vPx
namespace {
    const int MAX_A = 12;          // the same as MAX_B, but no static needed
}
static const int MAX_B = 10;      // in the global namespace
struct Data {
    static const int SIZE = 14;    // as a data field in a class
};
void func() {
    static const int LIMIT = 16;  // as a local constant
}
```

The distinction mainly arises because `static` has a different meaning in each of these cases:

- **const in an anonymous namespace—MAX_A**

First without `static`. The constant `MAX_A` is only known in this `*.cpp` file. You cannot access it from another file.

- **static in the global namespace—MAX_B**

Instead of an anonymous namespace, you can use `static` to define a constant. This has the same effect and is the older syntax. The disadvantage is that you cannot make classes local this way. Prefer an anonymous namespace.

- **static const as data field—SIZE**

The `static` ensures that all variables of type `Data` share a single `SIZE`. The `const` makes it immutable also.

- **static const as local variable—LIMIT**

For a local variable, `static` means that it retains its value when the function is exited. The variable is initialized only on the first pass. Together with `const`, it becomes a constant that is initialized only when needed.

13.10.6 Even More Constant with “constexpr”

In some cases, the compiler requires you to use a *constant expression*—one that it can evaluate at compile time. This happens, for example, when you want to create an array of a certain size, as in the following listing.

```
// https://godbolt.org/z/E6MWxY1f7
#include <array>
int main() {

    std::array<int, 5> arr5{};    // literal and thus a constant expression
    std::array<int, 2+3> arr23{}; // 2+3 can be evaluated by the compiler

    const size_t SIZE = 5;      // defines a constant
    std::array<int, SIZE> arrSC{}; // can often be used by the compiler

    size_t size = 7;
    std::array<int, size> arrVar{}; // you cannot use a variable
}
```

Listing 13.12 Some expressions must be known at compile time.

For `SIZE`, I define a constant with `const`, and then I can use it as a constant expression. A variable like `size` is not possible at this point—nor in some other places.

So far, so good. Can you use anything that is constant as an array size? Unfortunately, no. There are cases where it is not enough to mark something with `const`. Because the actual semantics of `const` say, “This must not be changed at runtime.” This is a big

difference from saying, “The compiler must be able to calculate it at compile time,” even though they often coincide.

The following listing shows an example in which it the constant goes wrong.

```
// https://godbolt.org/z/xPzcWPdMa
#include <array>
struct Data {
    static const size_t LATE;           // declare constant
    static const size_t EARLY;         // declare constant
};

void func() {
    int x = Data::LATE;                // use constant
}

const size_t Data::EARLY = 10;        // define constant

std::array<int, Data::EARLY> arrEARLY {}; // use constant
std::array<int, Data::LATE> arrLATE {};  // ⚡ use constant
const size_t Data::LATE = 10;          // define constant

int main() {
    func();
}
```

Listing 13.13 Whether the compiler can use a constant in a constant expression is not always immediately apparent.

Here, the `EARLY` and `LATE` constants are defined in the `Data` class. `static` here means that all `Data` variables share a common constant or variable. As you already know, I first made the definitions outside the class for `EARLY` and `LATE`; within `Data`, only the “space” for them is defined.

However, because the `arrLATE` definition comes after its usage, the—seemingly—constant expression cannot yet be evaluated by the compiler here. And although the `int x = LATE; in func()` appears in the program code before the definition, it is not a problem here: the simple initialization of a variable does not need to be computed by the compiler in advance; it is sufficient that it can be done at runtime.

This “reverse order” is not always as clear as in this simple example. Definitions and declarations can indeed be in different `*.cpp` files. You can also write `const` before many more things than the compiler can compute in advance. For example, you can write and use

```
const double sin0 = sin(0.0);,
```

but the compiler cannot compute sine at translation time.

Is this all quite confusing? I agree with you. Therefore, I recommend declaring constants of this kind not only with `const`, but with `constexpr`—an enhancement of it.

For values you declare with `constexpr`, the compiler checks if it can compute them at compile time. If that is not the case, you will receive an error message.

With `constexpr`, [Listing 13.13](#) looks like the following listing.

```
struct Data {
    static constexpr size_t LATE; // ⚡ does not work without direct initialization
    static constexpr size_t EARLY = 10;
};
constexpr size_t Data::LATE = 10; // ⚡ with constexpr, definition is different
```

Listing 13.14 With “constexpr”, the compiler sees when an expression is not computable early.

You cannot declare `LATE` without also defining it—that is, initializing it at the same time. This ensures that you can use the constant as soon as its name is known.

You can use `constexpr` variables almost anywhere you can use `const`. Try to get used to using `constexpr` for these purposes.

“constexpr” as Return Type

Another exciting feature of `constexpr` is that you can also mark functions with it. Write `constexpr` instead of `const` for the return type. In fact, this `constexpr` affects the entire function:

```
// https://godbolt.org/z/Yv85s9T6v
constexpr size_t doubleIfTooSmall1(size_t value) {
    return value < 100 ? value*2 : value; // returns double if less than 100
}
std::array<int, doubleIfTooSmall1(50)> arr {};
```

This is huge progress! A self-written function that simply returns a `const` value cannot be used as a constant expression:

```
const size_t doubleIfTooSmall2(size_t value) {
    return value < 100 ? value*2 : value;
}
std::array<int, doubleIfTooSmall2(50)> arr {}; // ⚡ not constant enough
```

Why is there a distinction being made again here? The reason is so that the compiler ensures that it can calculate `doubleIfTooSmall1(50)` at compile time.

Here are two examples of what the compiler does not allow before C++23:

```
#include <cmath> // sin, cos, sqrt
constexpr unsigned myLog2(unsigned number) { // ⚡ function has static variable
    static int count_calls = 0;                // ⚡ not allowed in constexpr
    count_calls += 1;
    unsigned count = 0;
    while(number > 1) { number /= 2; }
    return count;
}
constexpr double one(double x) {               // ⚡ uses non-constexpr
    return sqrt(sin(x)*sin(x) + cos(x)*cos(x));
}
```

Since C++14 and especially since C++20 and C++23, the rules for what can be in a constexpr function are much less strict. For example, since C++14, simple while and for loops are allowed. Since C++20, you can even allocate memory with new within them. Since C++23, the static local variable is allowed. This in turn has allowed many methods and constructors like vector and string to be defined as constexpr, so you can now use them in constexpr functions—but only if their memory is also freed at compile time. Many algorithms in <algorithm>, <numeric>, and <utility> are now also constexpr.

Conveniently, constexpr implies inline for functions and member variables and const for member functions.

13.10.7 “if constexpr” for Compile-Time Decisions

Sometimes you want to decide which code to execute at compile time rather than at runtime. You do this all the time with overloads. “If the argument is a pointer, then dereference it; otherwise, return the value itself.” This problem could be solved with an overload, but it can also be done with if constexpr.

```
// https://godbolt.org/z/zMbnYzo17
template<typename T>
auto deref(T t) {
    if constexpr (std::is_pointer_v<T>) {
        return *t;
    } else {
        return t;
    }
}
int main() {
    int i = 42;
    std::cout << deref(i) << '\n';           // directly the value
```

```

    auto p = std::make_unique<int>(73);
    std::cout << deref(p.get()) << '\n';    // dereferenced pointer
}

```

Listing 13.15 With “if constexpr”, you can decide at compile time what code will be executed.

Here it is decided at compile time what code will be executed. For `I`, it is `return t—` because `is_pointer_v<int>` is false. For `p.get()`, it is `return *t`.

Overloads, or template specializations, can also be replaced with `if constexpr`.

```

// https://godbolt.org/z/eo468MvMo
#include <iostream>
#include <string>
struct S {
    int n;
    std::string s;
    float d;
};
template <std::size_t N> auto& get(S& s) {
    if constexpr (N == 0) return s.n;
    else if constexpr (N == 1) return s.s;
    else if constexpr (N == 2) return s.d;
}
int main() {
    S obj { 0, "hello", 10.0f };
    std::cout << get<0>(obj) << ", " << get<1>(obj) << "\n"; // Output: 0, hello
}

```

Listing 13.16 “if constexpr” also works with “else”.

Before `constexpr if`, you needed to write three template specializations, like `template<> auto& get<0>(S& s)`. The new approach can be more concise.

With this feature, you can very easily calculate Fibonacci numbers—a favorite example of this book—at compile time:

```

// https://godbolt.org/z/44KMf1fWr
template<auto N>
constexpr auto fibonacci() {
    if constexpr (N>=2) {
        return fibonacci<N-1>() + fibonacci<N-2>();
    } else {
        return N;
    }
}

```

```
int main() {
    std::cout << fibonacci<10>() << '\n'; // Output: 55
    std::cout << fibonacci<20>() << '\n'; // Output: 6765
}
```

However, because this is written as a nasty recursion, it may, depending on the implementation, cause the compiler to hang. But maybe you want to test the compiler's performance?

13.10.8 C++20: “consteval”

Since C++20, there has been a superlative of `constexpr`. If you declare a function with `consteval`, you have an *immediate function*. The compiler then executes the function code at compile time and inserts the result at the call site.

```
// https://godbolt.org/z/v3x7nGs3f
int get_input() {
    return 50; // or read something from a file or so
}
constexpr auto calculate_1(int input) {
    return input * 2;
}
consteval auto calculate_2(int input) {
    return input * 2;
}
int main() {
    int input = get_input();
    auto a = calculate_1(77); // at compile time ... maybe computable
    auto b = calculate_1(input); // ... computable, but valid
    auto c = calculate_2(77); // ... computable
    auto d = calculate_2(input); // ✗ ... not computable, invalid
}
```

Listing 13.17 With “consteval”, you can execute functions at compile time.

While `constexpr` merely offers the *possibility* to be computed at compile time, `consteval` is explicitly intended for that purpose. A `consteval` function can only be called with constant expressions. There is a good chance that the compiler will actually use the result at that point and the function will not need to be called at runtime.

Because `calculate_2(input)` depends on a variable whose value is not known at compile time, the compiler does not allow the function call.

Therefore, a `consteval` function can only perform `constexpr` computations. Such a function can be quite complex in itself (you can follow the Compiler Explorer link in the

next listing to see the machine code and confirm that the calculated primes are part of the compiled program).

```
// https://godbolt.org/z/Wbr7n8M4q
#include <iostream>
#include <array>
constexpr bool isPrime(int n) { // computable at compile-time
    if(n < 2) return false; // 0,1 are not prime
    for (int i = 2; i*i <= n; i += i>2 ? 2 : 1) { // 2,3,5,7,9,11,13,15...
        if (n % i == 0) return false;
    }
    return n > 1; // for 0 and 1
}
template<int Num>
constexpr std::array<int, Num> primeNumbers() { // only at compile-time
    std::array<int, Num> primes{};
    int idx = 0;
    for (int val = 1; idx < Num; ++val) {
        if (isPrime(val)) primes[idx++] = val;
    }
    return primes;
}
int main() {
    // initialize with prime numbers
    auto primes = primeNumbers<100>(); // 1000000 doesn't work
    for (auto v : primes) {
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```

Listing 13.18 Even more complex functions can be computed at compile time with “constexpr”.

Here, the array with the first prime numbers is computed at compile time. It may be tempting to shift things that are computationally expensive at runtime to compile time, but it is not always a good idea. A compiler typically has a limit for the expressions it can compute at compile time (the standard guarantees about one million). When I try to compute the first 1,000,000 prime numbers with my g++ 13.2 setup, I get this error:

```
'constexpr' evaluation operation count exceeds limit of 33554432
(use '-fconstexpr-ops-limit=' to increase the limit)
```


So, you should only shift such initialization work to translation time to a limited extent.

13.10.9 “if consteval”

Starting from C++23, there is also *if consteval*. This feature works a little differently than the normal *if* as it does not contain a condition in parentheses. Instead, the block following *if consteval* is only executed if the calling context is *constexpr* (which usually means it can be evaluated at compile time).

```
// https://godbolt.org/z/7KWqTPo1K
#include <iostream> // cout
constexpr int get_value() {
    if consteval {
        return 42;
    } else {
        return 668;
    }
}

int main() {
    auto a = get_value();
    std::cout << a << '\n';          // Output: 668
    constexpr auto b = get_value();
    std::cout << b << '\n';          // Output: 42
}
```

Listing 13.19 The compiler determines whether the call can be evaluated at compile time in the given context.

Although *a* is not evaluated in a *constexpr* context, *if consteval* detects that the caller is not constant and inserts the *else* branch at this point during compilation. For *b*, however, the *if* branch is used because *b* is *constexpr* and can be evaluated at compile time.

This is just another way of writing the following:

```
constexpr int get_value() {
    if(std::is_constant_evaluated()) {
        return 42;
    } else {
        return 668;
    }
}
```

Because this was often confused with *if constexpr*, the new syntax was introduced.

13.10.10 Un-“const” with “mutable”

Psssst! Here’s a secret for you, and you must not tell anyone. And don’t use it, okay? You can make a field *mutable* in a class, *even if* you are currently in a `const` method.

By doing this, you bypass the assurance that the compiler provides you. But sometimes you have to make an exception. For example, if you want to count the number of calls to a method for analysis purposes, which is actually `const`.

```
// https://godbolt.org/z/sPorKnEbd
#include <iostream>
class Data {
    int value_;
    mutable size_t getCount_{0};
public:
    explicit Data(int v) : value_{v} { }
    ~Data() {
        std::cout << "get was used " << getCount_ << " times\n";
    }
    int get() const {
        ++getCount_;
        return value_;
    }
};
int main() {
    Data d{42};
    for(int i=0; i<10; ++i) { d.get(); }
} // Output: get was used 10 times
```

Listing 13.20 “mutable” makes a data field in `const` methods mutable.

If you were to omit `mutable`, the `++getCount_` expression in `get()` would be an error. In methods marked with `const`, it is not possible to modify instance data fields. With `mutable`, you make an exception for a single field.

And you should treat `mutable` as an exception: use it only for analysis. If you need it for another reason, reconsider your design.

You can also append a `mutable` to a lambda. I describe this in more detail in [Chapter 23, Section 23.3.6](#) under the “Mutable Lambdas” section.

13.10.11 Const-Correctness

A related and very helpful aspect in the long run is to get used to the correct use of `const`. Coming from C (or Java), this probably means using more of it. Once in the C++

world, the compiler can already uncover many *logical errors* in the code with `const` in the right places—where something was changed that was not meant to be changed.

The second aspect is documentation. A `const` tells subsequent programmers (all too often yourself) what this value is intended for. Is the variable used for computation, or just for temporary storage? Is it an input parameter or an output parameter?

Iterators for Constants

In addition, there are other places where the immutability of a value can be indicated. I will go into iterators in more detail in [Chapter 20](#), but as they are somewhat constant, I should explain them here.

A `Container::const_iterator` is a reference into a standard library container that does not modify the elements within. It should be considered similar to an `int const*`: the *value* cannot be changed, but the *reference* can. Thus, it is different from `const Container::iterator`, which is comparable to an `int *const`. The iterator itself remains fixed.

```
// https://godbolt.org/z/GE5v6Ea8x
#include <map>
#include <string>
using std::map; using std::string;
struct MyClass {
    bool isFound(const map<int,string> &dict, // immutable input parameter.
                const int &key,             // likewise
                string &result              // output parameter: not const
                ) const                      // instance of MyClass const
    {
        const map<int,string>::const_iterator where // reference and value fixed
            = dict.find(key);
        if(where == end(dict)) {
            return false;
        } else {
            result = where->second;
            return true;
        }
    }
};
```

Listing 13.21 “const” with containers.

The `find` method returns an iterator. However, it can be implicitly converted to a `const_iterator`. The other way around would not work implicitly.

13.10.12 Summary

If you equip your program with as many `consts` as possible—or necessary—from the beginning, then the compiler can detect many typos, logical errors, and sometimes even design errors during development.¹

The interface of classes and functions is described rudimentarily with `const` but without explicit documentation. In the program code, your future audience will thank you because `const` helps them understand what was meant to be changed and what was not.

Making a project *const-correct* from the beginning takes far less effort than having to do it later for debugging purposes.

The correct use of `const` is a powerful, not to be underestimated tool. It is better to start with “many” `consts` and consider whether you can omit them at a certain point rather than consider whether you should add more.

Using `const` correctly is part of type-safe programming. C++ has a powerful type system for error prevention. Use C++'s capabilities to operate on types at compile time and to compute with them.

13.11 Volatile with “volatile”

The `const` and `volatile` keywords are often mentioned together. There are two reasons for this: first, they form the common group of *cv-qualifiers* in the standard; and second, `volatile` is somewhat the opposite of `const`. With `volatile`, you mark things that can be changed particularly easily. This refers to the control of the compiler: with `volatile`, you mark those variables and fields that can change at any time from the perspective of the processor and compiler without their intervention—through some other external mechanisms.

C++20: Fewer “volatile”

In C++20, `volatile` was significantly pruned. The “dubious and defective parts” were removed, while the “useful ones remained,” as J. F. Bastien stated in a lecture.²

Historically, `volatile` can appear anywhere `const` can appear. This is nonsensical and confuses both programmers and compiler builders alike. Just because a parameter can be `const`, this was also allowed for `volatile`, which is quite pointless or at least confusing:

```
int func(volatile int x); // ⚡ volatile is pointless here
```

¹ “Item 43: Const-Correctness”, Herb Sutter, *Exceptional C++*, Addison-Wesley 2000

² “Deprecating volatile”, J. F. Bastien, *CppCon 2019*, https://www.youtube.com/watch?v=KJW_DLaVXIY

Since C++20, `volatile` has been decoupled from `const` and given a clearer semantics. By and large, `volatile` variables still serve the purpose of interacting with the external world, but the tricky cases are removed.

For example, a `volatile` variable could be a variable that reflects the state of a connected device. In essence, and extremely simplified, it could look like the following listing.

```
#include <iostream>
#include <chrono> // milliseconds
#include <thread> // this_thread::sleep

struct Pos {
    volatile int x;
    volatile int y;
};
Pos pos{};

// defined somewhere else:
int installMouseDriver(Pos *p);

constexpr auto DELAY = std::chrono::milliseconds(100);
constexpr auto LOOPS = 30; // 30*100ms = 3s
int main() {
    installMouseDriver( &pos ); // MouseDriver updates x and y
    for(int i=0; i<LOOPS; ++i) {
        std::cout << "mouse at ("<<pos.x<<","<<pos.y<<")\n";
        std::this_thread::sleep_for(DELAY);
    }
}
```

Listing 13.22 `x` and `y` can somehow be changed from outside.

The `x` and `y` fields of `Pos` have been qualified with `volatile`. This tells the compiler that the values in these fields could change at any time. The compiler should neither apply optimizations nor copy these variables into a register for calculations, for example. All calculations must be done directly in the memory of these fluctuation variables.

As I suggest in this example, `volatile` is particularly common in low-level programming. For instance, if you want to read a temperature sensor on your Raspberry Pi via the *general-purpose input/output* (GPIO) interface, one way to do this is to access the GPIO pin through a `volatile` variable.³

³ <https://www.raspberrypi.org/forums/viewtopic.php?t=130798&p=875156>, [2024-02-25]

There are (especially since C++20) rules about what you should not do with a `volatile` variable:

- Do not use it as a synchronization mechanism between threads. There are other mechanisms that are better and safer for this purpose.
- Do not use compound assignments like `x += 1` or `v |= 0x03` on a `volatile` variable. It may—but does not have to—result in two accesses to the variable, which would be inconsistent. However, you may write `x = x+1` or `v = v | 0x03` as it is clear that there is one read and one write operation. Also, `++x` is allowed.

Chapter 14

Good Code, 3rd Dan: Testing

Professionalism must be pursued in software development in all phases: in planning, communication, programming (of course), testing, quality assurance, documentation, and, last but not least, maintenance.

The aspect of programming in C++ is the main focus of this book. In this chapter, I would like to introduce you to another part of this chain, which also has a lot to do with C++: testing.

Requirements for Tests

There are several requirements for tests themselves. These are some important criteria roughly in order of their importance:

1. Tests must be meaningful.
2. Good tests provide reproducible results.
3. They should be as automated as possible.
4. They should be able to be conducted regularly.

14.1 Types of Tests

To ensure the quality of your product, you will test it. This can happen in different ways and at different stages. They are all important, and depending on the project, more or fewer tests take place at one stage or another.

You will quickly notice that with the intention of wanting to test your code in mind, you will program differently. You will want to create interfaces that simplify *automatic testing* for you. Regular testing increases the quality and sustainability of your code. And if it is regular, then it should be easy to perform the test—ideally, automatically. Moreover, reproducibility is very important in tests. Because what good is it if a test fails and you cannot analyze why?

I would like to mention the following two testing types as representative of all other types of tests, as they represent the extremes:

■ Functional and system testing

At one end of the testing spectrum, your program will be tested in its entirety. Ensure during the design and programming phases that these tests are easy to

perform, reproducible, and meaningful. In practice, different flavors of these tests will occur. Here are some keywords, which by no means cover all types of tests:

- A *smoke test* quickly checks the minimum functionality.
- In a *data-driven test*, you check whether a predefined set of input data triggers the desired behavior of your software.
- With a *load test*, you determine the maximum load your software can handle.
- You compare memory or time consumption with a *performance test*.
- An *integration test* usually takes place in a separate environment that is as close as possible to the actual deployment of the software.
- In an *end-to-end test*, your program may be just one component of a long chain, primarily checking the communication paths between the elements.

■ Unit test

At the other end of the spectrum are tests of the smallest units of the program. In C++, these smallest units are typically functions and methods. Developers test these personally and pay less attention to the interaction of components (which is why unit tests complement good functional tests, not replace them!). Unit tests must be able to be executed quickly as they are usually performed several times a day. This way of testing has become established in many software development methodologies and has become part of the daily routine for many programmers.

To conduct functional and system tests in a structured manner, there are many tools and utilities available, but they are too diverse and too dependent on the field of application for me to have any chance of giving you a useful guide in this book. Moreover, they are less specific to C++ and therefore do not belong in this book.

Unit tests are different. In this chapter, you will find tips on how to automatically test your C++ program using simple tools.

14.1.1 Refactoring

Software that you want to further develop must be adapted over time. In doing so, you must also dare to touch on existing code sections. If you don't do this, you will end up with bad code over time: duplications, errors, outdated technology, and so on. Regular revision is the remedy for this.

When you revise your code without changing its functionality, this is called *refactoring*. For example, you reorganize it, rename variables, or add parameters. Often, this is preparation for a necessary extension: perhaps you want to make a currently specific function more general with an additional parameter, or you want to extract functionality that is implemented in multiple places into a single function and call it from those places.

Such changes are often necessary across large code areas, and it is difficult to foresee the consequences of a change. This can lead to hesitation in modifying your code, which in turn makes the necessary extension more difficult. Therefore, you must develop from the beginning in a way that allows for refactoring, and you achieve this best through testable code—for unit and functional tests. Without this combination, you cannot maintain high code quality in the long run.

Test-Driven Development

In the test-driven development (TDD) approach to programming, you first write the (unit) test and run it. Of course it fails, because you haven't written any program behind it yet. If it *doesn't* fail, you are doing something wrong because either your program already has the functionality (unlikely), or your test is useless and tests nothing. Only *then* do you extend your program with the functionality that the test checks. Do not program *more* than the test requires to avoid programming untested functionality. Developers must first think about the interface of the function they want to write from an external perspective. In C++, you create the interface, for example, using an empty function. This is followed by the test, which fails, then by filling the function until the test passes.

This type of programming requires a lot of practice and discipline, but eventually it becomes second nature. You increase the quality of your code by having tests for everything and inevitably thinking more about the interfaces, because your test is your first user.

14.1.2 Unit Tests

In fact, there is not *one* definition of unit tests; there are several. However, they all share the following aspects:

- **Low-level**

The tests occur at a very low level of the source code. In C++, these could be classes or individual methods and functions. However, this is just a rough classification, and often a class is checked by multiple tests, or groups of functions are tested together.

- **Authors program unit tests**

You write this type of test yourself. It is sensible and necessary for other people to test as well, but the unit test is written by the same person who writes the code being tested. As part of *refactoring*, tests can be extended, supplemented, or adjusted by other programmers.

- **Run through quickly**

Unit tests must be run several times a day. Therefore, you must not condemn the programmers to wait. Typically, the tests run with each compilation pass.

■ Not part of the productive code

Unit tests are *outside* the program. They should normally complement the productive code, not be part of it (see [Figure 14.1](#)).

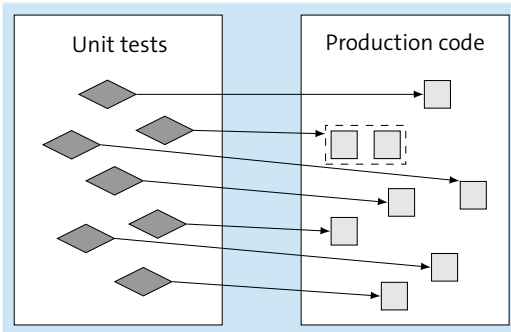


Figure 14.1 Unit tests are separate from the productive code and test individual units of it, occasionally multiple units. In solitary tests, these units are strictly separated.

In practice, you write your productive code in a certain way to make it easily testable—but the test itself is elsewhere. A data access object (DAO), which would normally be private and `const`, might be friend or protected for testability purposes and give up its write protection. It is better for the production code to get by with as few of these compromises as possible, but designing and writing the code to be *testable* is more important.

14.1.3 Social or Solitary

However, there is a significant point where practices differ. Isolate your components during testing to conduct *solitary tests*. But if you also check the collaboration of multiple components, then these are *social tests*. Which direction you lean toward will depend on you, your project, and your team. Both occur in practice.¹

In solitary tests, interaction with other components is intentionally kept low, if necessary, by substituting *mock objects* specifically for the test, which behave as if there were a real component (see [Figure 14.2](#)).

```
// https://godbolt.org/z/a83sf9h8E
#include <iostream>
// Production code:
struct DatabaseInterface {
    virtual int getData() const = 0;
};
struct Program {
    DatabaseInterface &db_;
```

¹ *Unit Test*, Martin Fowler, <http://martinfowler.com/bliki/UnitTest.html>, 2014-05-05, [2017-11-25]

```

    void run() {
        std::cout << db_.getData() << "\n";
    }
};
// Test helper:
struct MockDatabase : public DatabaseInterface {
    int getData() const override { return 5; }
};
// main as test:
int main() {
    MockDatabase mockDb;
    Program prog { mockDb }; // real DB is not tested
    prog.run();              // Expected output: 5
}

```

Listing 14.1 Solitary tests test only one component, and helper classes are substituted.

Social tests, on the other hand, aim precisely at the interaction of the components and check whether the connections work as intended.

```

// https://godbolt.org/z/fsroE4seh
#include <iostream>

// Production code:
struct DatabaseInterface {
    virtual int getData() const = 0;
};
struct RealDatabase : public DatabaseInterface {
    int getData() const override { return 999; }
};
struct Program {
    DatabaseInterface &db_;
    void run() {
        std::cout << db_.getData() << "\n";
    }
};

// main as test:
int main() {
    RealDatabase db;
    Program prog { db }; // real DB is tested along
    prog.run();          // Expected output: 999
}

```

Listing 14.2 Social tests test the interaction of components.

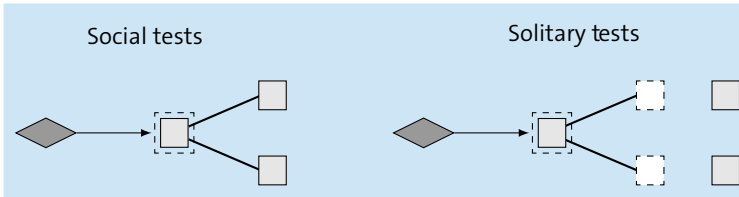


Figure 14.2 Social tests include connections to dependencies. Solitary tests decouple connections with helper classes.

As a rule of thumb, in a solitary test during the test run, the following applies:

- Only use one concretely instantiated class during the test.²
- Do not cross any boundaries.

A *boundary* in this sense is “a database, a queue, another system, or even just another class that is outside the range you are currently working on or responsible for.”³

14.1.4 Doppelgangers

When testing a unit, you often face two particularly significant challenges:

■ Indirect inputs

For example, if the component uses global variables, it makes testing more difficult. You need to at least be aware of them, which requires good documentation or a careful look at the source code. In addition, the global variable can be altered from elsewhere during the test, which does not make testing easier. You need to somehow deal with the problem of *shared states*. You don't even have to go global to encounter problems; often, a field of the tested object that is part of the shared states is enough.

■ Indirect outputs

It might not be as problematic during the test if the test element changes things that remain invisible to the tester. However, for the overall behavior of the software, it does make a difference what else happens—what *side effects* the tested code has.

One way to control both things is *test doubles*. They are used to isolate the shared state and to capture the side effects. There are different types of these test doubles, which can be roughly divided into the following categories:

■ Dummy

A *dummy* is never called and only serves to fill a spot that would otherwise cause problems during the test—for example, if the function under test checks for the

² Value parameters do not count here and are okay.

³ *TDD Pattern: Do Not Cross Boundaries*, William E. Caputo, <http://www.williamcaputo.com/posts/000019.html>, 2004-12-31, [2024-08-11]

presence of a certain parameter, but this parameter otherwise has no effect during the test.

- **Fake**

A *fake* is called but does the bare minimum to keep the test running—for example, `getName()` to see outputs during the test.

- **Stub**

A *stub* does a bit more than a fake as it provides the indirect inputs that the test object needs during the test. For example, if the test function reads from a stream, a stub would provide this stream.

- **Spy**

With a *spy*, you store the indirect outputs to check them for correctness within the test function.

- **Mock**

A *mock* goes the furthest as it simulates more complex behavior within the test. Thus, it mainly provides indirect inputs, but often also needs to store things itself to bring them into the correct order for the test.

A mock and a spy in particular are sometimes difficult to distinguish from each other. For example, if you have already written an elaborate mock that reacts differently to various states, you can also build in a correctness check of the test results. And different authors provide different definitions for a fake and a stub as well. For instance, Fowler defines a fake as “a working implementation with some shortcuts.”⁴

Many test frameworks come with tools to create these doubles during tests and simplify their handling. How they do this and what they provide depends on the framework.

Nevertheless, writing good tests in larger projects is not easy. Both solitary and social tests can prove to be tricky in practice. In social tests, this is often due to their inherent dependencies, which can be more or less extensive. If writing a solitary test is particularly difficult, it usually reveals deficiencies in the design of the code being tested—or, as they say, “Solitary unit testing makes bad OOP stinky.”⁵

14.1.5 Suites

Regarding the speed of unit tests, there are also differences. One person feels secure with a test suite that takes a minute to run. Another says a wait time of more than 300 milliseconds is too much.

4 *Mocks Aren't Stubs*, Martin Fowler, <https://martinfowler.com/articles/mocksArentStubs.html>, 2007, [2017-11-25]

5 *Principles of Unit Testing*, Joseph D. Purcell, <https://josephdpurcell.github.io/principles-of-unit-testing/presentations/2017-03-17-MidwestPHP/index.html#/6/17>, at Midwest PHP March 2017, [2024-05-19]

To accommodate both, it is common to set up (at least) two *suites* of unit tests in your project:

- **Compile suite**

Executed with every compile operation and should therefore run very quickly. As a rule of thumb, the test duration should be in the same order of magnitude as the compilation.

- **Commit suite**

Performing a *commit* is completing a task by sending the source code back to the team. Once or twice a day is typical; in extreme cases, it might happen once a week. To ensure that everything still works as desired, you should allow this suite a bit more time to run—for example, a few minutes. However, it shouldn't be much longer than that because if there are errors to fix, you need to be able to run the suite multiple times.

14.2 Frameworks

It is best to use a framework to write and execute your unit tests. There are different implementations for different languages, including C++. At the end of this chapter, I will introduce you to an example framework.

Certain terms have become established across frameworks. These are found as standard procedures grouped under the term *xUnit*. The following explanations are related in [Figure 14.3](#):

- **Test**

A test is a special function that is called by the framework. Everything in unit testing revolves around tests, because a single test tests a single unit, a functionality. Within a test, you call the function to be tested and then write one or more assertions to verify the correct functionality.

- **Assertion**

One assertion is performing a single check, typically the return value of the function to be tested. It always consists of a Boolean test, sometimes with a descriptive message upon failure. Often, side conditions are also checked with assertions. Some frameworks support two levels of assertions: warning and error. In the former, only a message is displayed if the assertion fails, while in the latter, the test fails. Here again, there are two variants: the test either continues or aborts.

- **Test case**

Multiple tests are combined into a test case. Sometimes a test case contains only one test (as a test function), but often there are several. These then have something in common; for example, they require the same test environment or the same test fixture.

■ Test fixture or fixture

Tests must be able to be considered in isolation from each other. Therefore, it is assumed for each test that it will again find a clean, unchanged environment. For this purpose, a test case has *one* special function that is called before *each* test within the test case, and *another* one that is called *after* each test. In these *setup* and *tear-down* functions, you set up the test environment and undo it again.

■ Test suite

A suite includes a set of test cases. You can define multiple suites and pack different test cases into each to perform, for example, various thorough tests.

The term *test* is not “official” here; in test-driven development and the family of frameworks summarized under the name xUnit, only *test case* is used.⁶ In practice, however, I find it easier to introduce a distinction in my mind. A test case should, one way or another, test *a* functionality, even if this examination is divided into multiple tests (technically, test functions).

When you run unit tests, the framework proceeds as follows: You select one or more suites to be executed. For all the test cases contained therein, all tests are gathered and defined as the set of tests to be executed. These are then all executed.

Each individual test can either pass (test is green) or fail (test is red). There may be non-critical warnings. If all tests were successful, this indicates overall success. A single failed test means a failure. It is not common to accept a certain number of individual errors.

Tests must run independently of each other. They must not rely on certain other tests having run before or not yet run. The framework can also run multiple tests simultaneously.

When executing a single test, the following happens:

1. The framework instantiates the test case containing the test.
2. Any existing setup function or method of the test case is called.
3. The framework calls the test using its test function.
4. The failure of an assertion or the unexpected exit of the test is noted and later reported as a fail.
5. If a teardown function or method exists, the framework calls it.
6. The test case is removed.

Note that this happens for *every* test: even if multiple tests are within a single test case, the test case is always set up and torn down anew for each of these tests.

⁶ *Test Driven Development: By Example*, Kent Beck, Addison-Wesley Longman (2002)

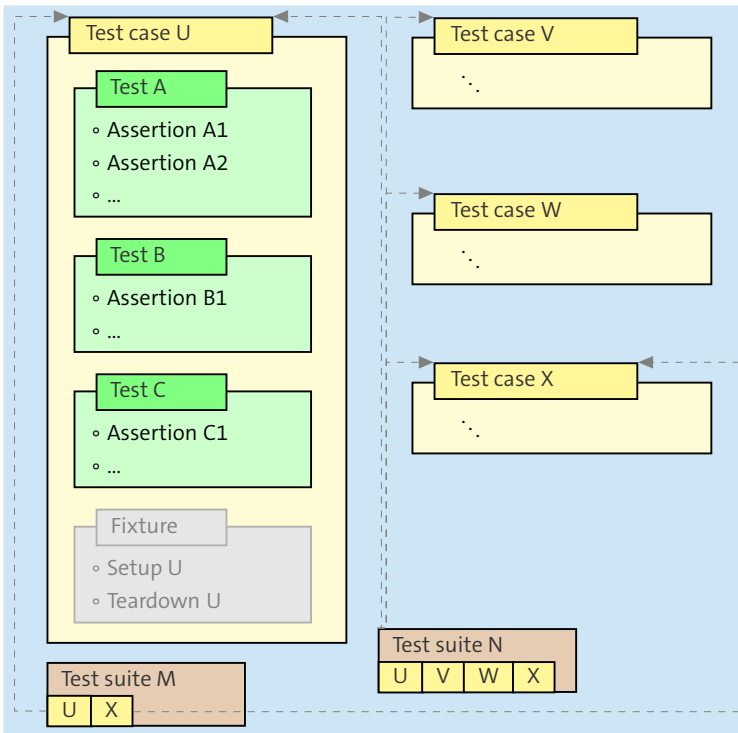


Figure 14.3 Test cases contain test functions with assertions and are assembled into test suites.

14.2.1 Arrange, Act, Assert

You will quickly notice that certain patterns always repeat when writing a test. I always comment within my test functions in three sections:

```
// https://godbolt.org/z/sTaoeqfxr
void testDoubleIt5() {
    // arrange
    auto param = 5;
    // act
    auto result = doubleIt(param);
    // assert
    assertTrue(result == 10);
}
```

Sometimes arrange is empty because I could have just written `doubleIt(5)`. However, the other sections are always filled; without them, a test makes no sense. This principle is called *arrange, act, assert*.⁷

⁷ <http://c2.com/cgi/wiki?ArrangeActAssert>, [2017-11-25]

Afterward, it is advisable that you also divide your code to be tested cleanly into these three sections. This makes writing and understanding easier. In the preceding example, this may seem ridiculously trivial, but if you first have to create a landscape of spy and mock objects before execution, a clean separation is helpful:

- In `arrange`, you declare and initialize variables and prepare the things needed during the actual test.
- Then, in `act`, you call the program code that is to be tested.
- In the `assert` part, all checks of the expected versus the actual result are made.⁸

As a rule, each of your tests should follow this pattern. In exceptional cases, you can combine all three sections into one line. Then, and only then, can you also call the code to be tested multiple times in a test routine—and that is my recommendation. It looks like this:

```
// https://godbolt.org/z/rn5oK3M6e
void testDoubleIt() {
    assertTrue( doubleIt(0) == 0 );
    assertTrue( doubleIt(-1) == -2 );
    assertTrue( doubleIt(1) == 2 );
    assertTrue( doubleIt(5) == 10 );
}
```

Test Corner Cases

Your tests should particularly, but not exclusively, cover the *corner cases*, such as the call with parameters 0 and the negative number. A good mix of typical calls is also important, here for example with 1 and 5.

For integers, 0 and -1 are often good values for edge cases, strings can be empty (""), and C strings and other pointers can be `nullptr`. Test your functions on various combinations of normal and corner cases.

14.2.2 Frameworks to Choose From

There are many implementations of test frameworks. Some support only unit tests, while others are good for other types of tests. Here is a small selection with a focus on unit tests and portability, but the complete selection is much larger:⁹

⁸ In some test frameworks, it is helpful to have only a single assert per test function, as it can be difficult to pinpoint the failed test. This is not the case with most frameworks, including the C++ frameworks shown here.

⁹ *Exploring the C++ Unit Testing Framework Jungle*, Noel Llopis, <http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle>, 2010, [2017-11-25]

- **Boost.Test**

This test framework is part of the Boost library collection. It is possible to use the library both as a header-only version and as a library to link against. I use it here to explain the examples.

- **C++ Unit Testing Easier (CUTE)**

A simple yet advanced unit test framework that can be integrated into Eclipse with a plug-in.¹⁰

- **CppUnit**

This is a fairly widespread unit test framework that is heavily inspired by JUnit.¹¹

- **doctest**

This is a small header-only library.¹²

- **Google Test**

Google also develops with C++ and has made its test framework available.¹³

14.3 Boost.Test

Let's dive right in and look at a program we want to test. The framework we are using here is Boost.Test.

Installing Boost.Test

You can download Boost as a complete package from the Boost website at <http://www.boost.org/users/download>. For Windows and MSVC, there are precompiled libraries available. On Ubuntu, `sudo apt-get install libboost-dev` installs the complete library. Other distributions have a corresponding package. It is not sufficient to install only the sublibrary of the test framework as it also uses other parts of Boost.

The following listing shows a free function that you just wrote.

```
// https://godbolt.org/z/Ex6Mv6xcz
unsigned fib(unsigned n) {
    if(n==0) { return 0; }
    if(n==1) { return 1; }
    unsigned a = 0;
    unsigned b = 1;
    unsigned sum = 1;
    while(n>1) {
        sum += a;
```

¹⁰ C++ Unit Testing Easier, <http://cute-test.com/>, [2017-11-25]

¹¹ CppUnit, <https://freedesktop.org/wiki/Software/cppunit/>, [2024-05-19]

¹² doctest, <https://github.com/doctest/doctest>, [2024-05-19]

¹³ Google Test, <https://github.com/google/googletest>, [2024-05-19]

```

        a = b;
        b = sum;
        n -= 1;
    }
    return sum;
}

```

Listing 14.3 In the “fib.cpp” file, there is a free function that you want to test.

This should be the entire content of `fib.cpp`. You write the tests in a separate module, separate from the production code—for example, in a separate directory like `test/`, or with a clear naming scheme such as `test_fib.cpp`.

To test the `fib` function in another module, you need to call it—and that requires that it is known to the test code. Either you need a header that exports the function and that you include in the test code, or you use your internal developer knowledge about the function and declare it before the tests without `#include`. It depends on your test rules whether you allow this as an exception and for unit tests, because in productive code such opaque use of functions is not good style. One might argue that unit tests are an exception here, because the naming scheme of the filenames indicates where the actual function is located.

Thus, the `test_fib.cpp` test file begins with the `unsigned fib(unsigned n);` declaration. The entire file then looks like this:

```

// https://godbolt.org/z/erPdKo1qh (uses boost library)
#define BOOST_TEST_MAIN test_main           // generates main() in this module
#include <boost/test/included/unit_test.hpp> // framework
#include <boost/test/test_tools.hpp>         // BOOST_CHECK etc
unsigned fib(unsigned n);                   // to test
namespace {
    using namespace boost::unit_test;
    BOOST_AUTO_TEST_CASE( test_fib_low )    // arbitrary name of the test case
    {
        BOOST_CHECK( fib(0) == 0 );         // individual assertions
        BOOST_CHECK( fib(1) == 1 );
        BOOST_CHECK( fib(2) == 1 );
        BOOST_CHECK( fib(3) == 2 );
        BOOST_CHECK( fib(4) == 3 );
        BOOST_CHECK( fib(5) == 5 );
        BOOST_CHECK( fib(6) == 8 );
    }
}

```

With the `BOOST_TEST_MAIN` macro, you tell Boost.Test to automatically generate a `main()` function in this module that contains all the administrative overhead: assembling the

cases and suites, evaluating the command line parameters, executing the tests, and presenting the results.

The `BOOST_AUTO_TEST_CASE` macro automatically creates a new test case with the specified name and adds it to the default test suite that `main()` runs. The test case generated in this way contains only a single test function, the body of which now follows. Here I use `BOOST_CHECK` multiple times as the simplest helper macro. It only takes one parameter that can be converted to `bool`. If this results in `false` during the test run, the test fails. In addition to `BOOST_CHECK`, there are several other assertion helper macros: more on that shortly.

First Actual, then Expected

Note that in Boost.Test, you should mention the *actual* result first and the *expected* result last. With the macros in [Table 14.1](#), it doesn't make a big difference, but if you use `BOOST_TEST` from library version 1.59 or later, the framework's output is more detailed if you have the comparison operator as far to the right as possible:

```
BOOST_AUTO_TEST_CASE( expected_to_the_right ) {  
    int a = 13, b = 2;  
    BOOST_TEST(a % b == 7); // 'check a % b == c has failed [13 % 2 != 7]'  
    BOOST_TEST(7 == a % b); // ✗ 'check a == c % b has failed [7 != 1]'  
}
```

You compile the test program by compiling both modules into an executable program:

```
$ g++ fib.cpp test_fib.cpp -o test_fib.x
```

When you run it, you see no errors:

```
$ ./test_fib.x  
Running 1 test case...  
*** No errors detected
```

This version of the Boost.Test library consists only of headers and no static or dynamic library that you need to specify when compiling. The `<boost/test/included/unit_test.hpp>` include file contains a `main()` and all the helper functions for it—and is not a *header* in the usual sense. As such, the file should not contain any implementations, especially not those that prevent compiling if included multiple times in a project. And because `.../included/unit_test.hpp` contains a `main()`, it would appear multiple times in the program, which is not allowed. But in this case, this exception is very useful and simplifies usage and compiling.

If you do not want to use the header-only variant, there is also the option to add the Boost.Test `main()` and surrounding as a library. Replace the

```
#include <boost/test/included/unit_test.hpp>
```

line with

```
#include <boost/test/unit_test.hpp>
```

and then compile the program by adding the framework library *statically*:

```
$ g++ -static fib.cpp test_fib.cpp -o test_fib.x -lboost_unit_test_framework
```

You must do this with `-static` because the `boost_unit_test_framework` library contains—as already explained—`main()` as well. On Linux, it is not possible to create a runnable program that does not contain `main()` itself but wants to load it dynamically. (Therefore, the dynamic variant does not contain `(main)` and is only needed in the advanced case where you use an external test runner.)

The Boost.Test framework inserts a `main` function for you, but it can do even more. For example, you can specify a whole range of command line parameters that Boost.Test automatically evaluates. Select, for instance, the test cases to be executed and change the output format and quantity:

```
$ ./test_fib.x --run_test=test_fib_low --report_level=detailed \
    --log_level=all --catch_system_errors=no
```

```
Running 1 test case...
```

```
Entering test suite "Master Test Suite"
```

```
Entering test case "test_fib_low"
```

```
test_fib.cpp(28): info: check fib(0) == 0 passed
```

```
test_fib.cpp(29): info: check fib(1) == 1 passed
```

```
test_fib.cpp(30): info: check fib(2) == 1 passed
```

```
test_fib.cpp(31): info: check fib(3) == 2 passed
```

```
test_fib.cpp(32): info: check fib(4) == 3 passed
```

```
test_fib.cpp(33): info: check fib(5) == 5 passed
```

```
test_fib.cpp(34): info: check fib(6) == 8 passed
```

```
Leaving test case "test_fib_low"; testing time: 346mks
```

```
Leaving test suite "Master Test Suite"
```

```
Test suite "Master Test Suite" passed with:
```

```
  7 assertions out of 7 passed
```

```
  1 test case out of 1 passed
```

```
Test case "test_fib_low" passed with:
```

```
  7 assertions out of 7 passed
```

With `--report_level`, you can influence the amount of summary output, as seen in the last five lines. `--log_level` influences the output during test runs and ranges from nothing to errors to all with many gradations in between.

The `--catch_system_errors=no` parameter is useful when you start the test within an IDE. Normally, Boost.Test goes to great lengths to catch even the worst errors so that the remaining tests can be run. This can be a hindrance when debugging as the built-in

debugger does not notice the error. With this option, for example, Microsoft Visual Studio can jump directly to the point where an error occurred.

Call the test program with `--help` or `-?` to get a detailed list of parameters.

In the output, you will see that Boost.Test provides the summary for the *master test suite*. It is a specialty of this framework that suites and cases can be hierarchically nested like a tree. The root is always the master test suite. Most other xUnit frameworks do not have this nested structure but are limited to a set of test cases in a set of test suites.

14.4 Helper Macros for Assertions

You have encountered `BOOST_CHECK(b)`. This basic macro checks if its parameter is `true`. If not, it marks the test as failed, but the test continues to run. If you use `BOOST_REQUIRE(b)` instead, the test will abort on `false`, and the remaining assertions of this test will not be executed. With `BOOST_WARN(b)`, on the other hand, you will receive a message on `false`, but the test does not fail. These are the three *assertion levels* consistently supported by Boost.Test. All assertion helper macros are available in these three variants.

I use `BOOST_level_EQUAL(res, exp)` most frequently. Here I specify the actual result with `res` and the expected result with `exp`. The comparison using `==` is handled by the framework for me; it can provide better information in case of inequality:

```
std::string theString(std::string val) { return val; }
BOOST_CHECK_EQUAL( theString("result"), "result" );
BOOST_WARN_EQUAL( theString("result"), "reslt" ); // ⚡ Abort
```

The output of the second test is

```
warning in "test_theString": condition theString("result") == "reslt"
    is not satisfied [result != reslt]
```

On the one hand, the return value of `theString` is conveniently included in the message here, and on the other hand, it would have an effect if you had defined `operator==` for the two arguments.

If you do not want to use `==`, instead use `BOOST_level_BITWISE_EQUAL(exp, res)`, where the arguments are compared directly in memory.

Instead of equality using `...EQUAL`, you can also use `GE`, `GT`, `LE`, `LT`, and `NE` for `>=`, `>`, `<=`, `<`, and `!=` respectively.

You must exercise special caution when you want to compare floating-point numbers. A test for equality using `==` is out of the question. Instead, check if `exp` and `res` are close enough by providing a third parameter that specifies what “close enough” means:

```
double sqSumDif(double a, double b) {
    return a*a+b*b-a*b;
}
BOOST_CHECK_CLOSE( sqSumDif(4.1, 3.9), 16.03, 0.01 );
```

The output here is

```
info: difference{%} between sqSumDif(4.1, 3.9){16.029999999999998} and \
    16.03{16.030000000000001} doesn't exceed 0.01%
```

because the actual *imprecise* result deviates from the expected one, but it is within the specified tolerance of 0.01%. Here, you should specify an appropriate tolerance on a case-by-case basis. The 0.01% tolerance is probably quite coarse for such a simple calculation, but in complex calculations, the rounding errors of floating-point calculations can accumulate.

Why Macros?

A small note on the fact that so many macros are used here, even though I keep telling you to avoid macros. Well, no rule is without exception, and tests like these are one of them. Because without using macros here, it would not be possible for the compiler to provide you with useful information such as

```
info: check fib(0) == 0 passed
```

on the screen. Here, the entire passed expression was converted into a string (“stringified”) using # to print this information, which is especially useful in case of an error. This would not have been possible with any `constexpr`.

You can see the complete list of helper macros in [Table 14.1](#).

Macro BOOST_level...	Description
BOOST_level(b)	Is b true?
....EQUAL(e,r)	Is e == r true?
....BITWISE_EQUAL(e,r)	Is e bitwise identical to r?
....EQUAL_COLLECTIONS(...)	Element-wise begin/end(e)==begin/end(r).
....GE(e,r)	Is e >= r?
....GT(e,r)	Is e > r?
....LE(e,r)	Is e <= r?
....LT(e,r)	Is e < r?

Table 14.1 A brief overview of all assert macros in Boost.Test.

Macro BOOST_level...	Description
...NE(e,r)	Is $e \neq r$?
...PREDICATE(p,v1,v2,...)	Are $p(v1)$, $p(v2)$, and so on all true?
...CLOSE(e,r,t)	Is e close enough to r , in %?
...CLOSE_FRACTION(e,r,t)	Is e close enough to r ?
...SMALL(r,t)	Is r close to zero?
...NO_THROW(a)	The expression a must not throw an exception.
...THROW(a,ex)	The expression a must throw exception ex .
...EXCEPTION(a,ex,p)	Ditto— ex must also satisfy condition $p(...)$.
BOOST_level_MESSAGE(b,msg)	Outputs message msg on false.
BOOST_ERROR(msg)	Like ...MESSAGE(false,msg).
BOOST_FAIL(msg)	Aborts the current test.
BOOST_IS_DEFINED(sym)	True if the macro sym is defined.

Table 14.1 A brief overview of all assert macros in Boost.Test. (Cont.)

The Universal Macro BOOST_TEST

In the current Boost versions, there is the all-rounder `BOOST_TEST`. This macro takes a comparison as the first parameter and can optionally take *test manipulators* as additional parameters. This allows you to influence the precision or logging behavior.

Thus, the following tests

```
BOOST_REQUIRE_EQUAL(qw::version(), 1);
BOOST_REQUIRE_GT(qw::version(), 5);
BOOST_REQUIRE_CLOSE(a+0.1, b-0.8, 0.01));
```

perform the same task as the following:

```
BOOST_TEST(qw::version() == 1);
BOOST_TEST(qw::version() >= 5 );
BOOST_TEST(a+0.1 == b-0.8, boost::test_tools::tolerance(0.01));
```

With this macro, the tests look more uniform, and the optional manipulators allow for more combinations and influence on the test behavior.

Since 1.59 (Boost.Test v3), the macro has been available. Systems from around 2017 should be able to install a suitable Boost version.

14.5 An Example Project with Unit Tests

To show you how unit tests can be integrated into an actual project, I will demonstrate the structure of a complete project in this section. It consists of

- a library,
- an example program that uses this library,
- and a set of unit tests for the library.

The whole thing is embedded in a Makefile project, but it can be transferred to a project in your favorite IDE without much effort.

Previews and Focus of This Section

I am expanding the example project from [Chapter 11, Section 11.4](#) here. For the basic structure, refer to that section first.

I will touch on a few things about classes and their design, which we discuss in detail later in this book. As always in the *dan* chapters, the focus is not on that, but rather on showing you things that will be useful when you return to this chapter later. I will also mention the *Pimpl pattern* briefly, as it is established and not limited to the new C++.

I check the functionality of the library using additional source files with unit tests, which are not delivered to the customer. I list the additional files in the project in [Table 14.2](#).

Directory/File	Description
test/	The actual unit tests
- Makefile	Building the unit tests
- testQwort.cpp	Tests of the qwort class
- testImplMultimap.cpp	Tests of the helper class

Table 14.2 Additional files in the example project for unit tests.

The `make all` command first builds all files relevant to the project. Finally, the unit tests in the directory `test/` are also built.

Because I refer to it directly, I repeat the header file of the implementation here.

```
// https://godbolt.org/z/j93cYaWd4
#ifndef QWORT_H // Header guard
#define QWORT_H
#include <string>
#include <string_view>
#include <memory> // unique_ptr
```

```
namespace qw {    // Namespace of the library

    int version();

    namespace impl_multimap {
        class index_impl;
    }

    class index {
        using index_impl = impl_multimap::index_impl;
    public:
        index();
        ~index() noexcept; // needed for pimpl
        index(const index&) = delete;
        index(index&&) noexcept;
        index& operator=(const index&) = delete;
        index& operator=(index&&) = delete;
    public:
        void add(std::string_view arg);
        size_t size() const;
        std::string getBestMatch(std::string_view query) const;
    public:    // public for tests
        std::string normalize(std::string arg) const;
    private:
        std::unique_ptr<index_impl> pimpl;
    };
} // namespace qw
#endif // Header guard
```

Listing 14.4 The main header file of the `qwort.hpp` library.

14.5.1 Private and Public Testing

Internally, the two `add()` and `getBestMatch()` methods call `normalize()`. The purpose of this method is to simplify the index—for example, by storing only uppercase letters. This must then be done symmetrically both when storing the patterns and when making the search query. It would have been sufficient to make the `normalize()` method private or protected.

When you then write a unit test for the `index` class, you can consider the entire class as the unit to be tested. In this way, you only test the public interface and do not need tests for the private areas. This applies to `pimpl` in the example, where we simply assume that it is correctly initialized; the tests we write for the class as a unit test this circumstance.

But if core functionality was implemented in a private method, then it forms a unit on its own that is worth testing separately. This is a “soft” rule that you should discuss within your team. Here, it is the case for `normalize()`. Therefore, I’ve placed the “actually private, but public for tests” methods in a separate section and marked it with `public` and an appropriate comment. This is the simplest way to make private methods testable from the outside, but there are a handful of other ways. Each of them has its pros and cons, so I am only showing this one here.

In [Listing 14.5](#), you will see an excerpt from [Chapter 11](#), [Listing 11.3](#), where part of the interface is made public solely for the purpose of testability.

```
class index_impl {
// ...
public: // test interface
    vector<string> _qgramify(string_view n) const { return qgramify(n); }
    static size_t _q() { return Q; }
    static std::string _prefix() { return PREFIX; }
    static std::string _suffix() { return SUFFIX; }
};
```

Listing 14.5 Public interface for testing only.

14.5.2 An Automatic Test Module

In the first example of a unit test module, I will show you how to let the Boost.Test framework handle the process of combining individual tests into a suite. For this, it has a few practical macros.

The `testQwort.cpp` test module from [Listing 14.6](#) tests the `qwort.cpp` library module. It is advisable to adhere to a naming scheme for naming the test modules. I use `test...` as a prefix, but you can also append `...Test` as a suffix or be creative in other ways.

```
// https://godbolt.org/z/TacMfah8E (uses boost library)
#include "qwort/qwort.hpp" // under test
#define BOOST_TEST_MODULE qwort
#include <boost/test/included/unit_test.hpp>
using namespace boost::unit_test;
BOOST_AUTO_TEST_CASE( version_is_1 ) {
    BOOST_TEST(qw::version() == 1);
    // the following comparison should fail, but only produce a warning:
    BOOST_WARN_EQUAL(qw::version(), 2);    // ✗ not equal, but continues
}
BOOST_AUTO_TEST_CASE( init_size_0 ) {
    qw::index inst{};                // arrange
    auto sz = inst.size();            // act
    BOOST_TEST(sz == 0u);             // assert
}
```

```
BOOST_AUTO_TEST_CASE( add_size_1 ) {
    using namespace std::literals::string_literals;
    qw::index inst{};           // arrange
    inst.add("s");              // act
    BOOST_REQUIRE_EQUAL(inst.size(), 1u); // assert
}

BOOST_AUTO_TEST_CASE( normalize ) {
    using namespace std::literals::string_literals;
    qw::index inst{}; // arrange
    // acts and asserts; could also be in separate functions
    BOOST_CHECK_EQUAL(inst.normalize("a"s), "A"s);
    BOOST_CHECK_EQUAL(inst.normalize("Town"s), "TOWN"s);
    BOOST_CHECK_EQUAL(inst.normalize("White Space"s), "WHITE#SPACE"s);
    BOOST_CHECK_EQUAL(inst.normalize("!Sym-bol."s), "#SYM#BOL#"s);
}

BOOST_AUTO_TEST_CASE( move ) {
    qw::index inst{};
    qw::index other = std::move( inst );
    BOOST_CHECK_EQUAL(other.size(), 0u); // pimpl successfully moved?
}
```

Listing 14.6 This test module tests `qwort.cpp`.

I start the includes with the header file of the module being tested—in this case, `qwort/qwort.hpp`. This makes the components to be tested known within this file.

Next are the includes of the test framework—here only one, namely, `boost/test/included/unit_test.hpp`. With this include, you have the option to automatically generate a `main()` function and a master test suite. The framework does this if you define the `BOOST_TEST_MODULE` macro *before* the include. You also give the test module a meaningful name—here, `qwort`. Note that you are allowed to define this macro at most once per test program before the include. Logically, it ensures that a `main()` function is generated, and there can only be exactly one per program.

After the `using` directive, the test cases follow. Each one is introduced by the macro `BOOST_AUTO_TEST_CASE`, given a descriptive name as a parameter, and otherwise has the form of a free function. The macro ensures that this free function is registered under the descriptive name in the master test suite, so that it is executed when the test program is run.

Within the test functions, I use the usual arrange, act, assert (AAA) pattern. In the `assert` section, I use the Boost macros from [Table 14.1](#).

The tests here are not particularly complicated and are mostly self-explanatory. However, I would like to point out a few smaller things.

The `BOOST_CHECK_EQUAL` macro has two parameters, which are compared using the Boost internal `equal_impl` function. This then either uses an appropriate overload for the comparison or falls back to `==`, and the same for `int`, so that with `int` `result` and the

```
BOOST_CHECK_EQUAL(result, 42);
```

test, ultimately `result==42` is executed. But how does it work with `const char *res`, and what does it look like here?

```
BOOST_CHECK_EQUAL(res, "TEXT");
```

If the macro simply used `==` here, it would fail because only the two pointers would be compared, not the contents of the C strings. Fortunately, the framework is smart enough to use a function template with a specialization for two `const char*` parameters. This way, it works for all parameters of the check macros.

In my comparisons, however, I specifically used the `"s` suffix after `"TOWN"`s to create a string literal instead of a C string literal. For example, in

```
BOOST_CHECK_EQUAL(inst.normalize("Town"s), "TOWN"s);
```

the following would have been incorrect:

```
BOOST_CHECK_EQUAL(inst.normalize("Town"s), "TOWN");
```

Because then at runtime, the C string literal would first have to be converted into a string. However, I have now done this at compile time with `"TOWN"s` and ensured that the Boost internal specialization for comparing two `string` parameters is used instead of a `const char*` with a `string`. Although that would have worked as well, I find it logical not to require these conversions during tests if it is not absolutely necessary.

However, to use the `"s` suffix for literals, it must be stated somewhere:

```
using namespace std::literals::string_literals
```

Also, `using namespace std` would have worked, because `literals` and `string_literals` are inline namespaces.

For demonstration purposes, I used `BOOST_TEST` twice at the beginning to demonstrate the macro.

And because a test program that runs without complaints makes one skeptical whether anything happened at all, I deliberately included a failing test:

```
BOOST_WARN_EQUAL(qw::version(), 2);
```

Because `qw::version()` returns the value 1, you should receive a warning when running the tests here. You should, of course, correct this test before publishing (sending to your team).

14.5.3 Compile Test

In this way, you list all test cases for the module. You compile this source file together with the `qwort` library and get an executable program:

```
g++ -o testQwort.x testQwort.cpp -lqwort -I../include -L../src/lib
```

In this form, you do not need a Boost library because the `.../included/...` variant is “header-only” and does not need to be linked.

This is how you run the test program and then see the following output:

```
$ ./testQwort.x --report_level=short
Running 5 test cases...
Test suite "qwort" passed with:
    8 assertions out of 8 passed
    5 test cases out of 5 passed
```

All tests ran successfully. With the `--report_level=detailed --log_level=all` arguments, you get to see more information. You will then see each individual assertion executed, and the descriptive names of the test suite and test cases will also appear. I will only show you the beginning of the output here:

```
$ ./testQwort.x --report_level=detailed --log_level=all
Running 5 test cases...
Entering test module "qwort"
testQwort.cpp(16): Entering test case "version_is_1"
testQwort.cpp(17): info: check qw::version() == 1 has passed
testQwort.cpp(19): warning: in "version_is_1": condition qw::version() == 2 \
    is not satisfied [1 != 2]
testQwort.cpp(16): Leaving test case "version_is_1"; testing time: 100us
testQwort.cpp(22): Entering test case "init_size_0"
testQwort.cpp(25): info: check sz == 0u has passed
testQwort.cpp(22): Leaving test case "init_size_0"; testing time: 81us
...
```

14.5.4 Assemble the Test Suite Yourself

Because the private implementation of the index is somewhat longer than the public interface, the tests for it are also somewhat longer.

In addition, I worked here without the practical `BOOST_AUTO_TEST_CASE` macro, which takes care of assembling the main test suite. This time I wanted to have all tests as methods of a separate test class. However, with `Boost.Test`, this means you have to manually add the individual methods to a suite. I take the opportunity to demonstrate this to you here because this is exactly the way to build large and potentially nested test suites.

In [Listing 14.7](#), you can see the source code of the `testImplMultimap.cpp` test module, which in turn tests the `impl_multimap.cpp` library module.

```
// https://godbolt.org/z/r9sdhaazz (uses boost library)
/* private header from lib directory: */
#include "impl_multimap.hpp" // to test
// we define init_unit_test_suite() ourselves, so DO NOT set:
/* #define BOOST_TEST_MODULE qgram */
#include <boost/test/included/unit_test.hpp>
#include <memory> // shared ptr
#include <vector>
#include <string>
using namespace boost::unit_test;
using namespace std::literals::string_literals;
using std::string; using std::vector;
/* === A test class for the class under test === */
using UnderTest = qw::impl_multimap::index_impl;
class ImplMultimapTest { // class with test methods
public:
    void testConstants() {
        BOOST_REQUIRE_EQUAL(UnderTest::_prefix().length(), UnderTest::_q()-1);
        BOOST_REQUIRE_EQUAL(UnderTest::_suffix().length(), UnderTest::_q()-1);
        for(size_t i = 0; i < UnderTest::_q()-1; ++i) {
            BOOST_CHECK_EQUAL(UnderTest::_prefix()[i], '^');
            BOOST_CHECK_EQUAL(UnderTest::_suffix()[i], '$');
        }
        BOOST_TEST(UnderTest::_q() == 3u);
        BOOST_TEST(UnderTest::_prefix() == "^^"s);
        BOOST_TEST(UnderTest::_suffix() == "$$"s);
    }
    /* === qgramify === */
    void testQgramifyEmpty() {
        UnderTest inst{};
        auto result = inst._qgramify("s");
        vector<string> expected{"^^$"s, "^$$"s};
        BOOST_CHECK_EQUAL_COLLECTIONS(
            result.begin(), result.end(), expected.begin(), expected.end() );
    }
    void testQgramify1() {
        UnderTest inst{};
        auto result = inst._qgramify("a"s);
        vector<string> expected{"^^a"s, "^a$"s, "a$$"s};
        BOOST_CHECK_EQUAL_COLLECTIONS(
            result.begin(), result.end(), expected.begin(), expected.end() );
    }
}
```



```
void testQgramify2() {
    UnderTest inst{};
    auto result = inst._qgramify("ab"s);
    vector<string> expected{"^^a"s, "^ab"s, "ab$"s, "b$$"s};
    BOOST_CHECK_EQUAL_COLLECTIONS(
        result.begin(), result.end(), expected.begin(), expected.end() );
}

void testQgramify3() {
    UnderTest inst{};
    auto result = inst._qgramify("abc"s);
    vector<string> expected{"^^a"s, "^ab"s, "abc"s, "bc$"s, "c$$"s};
    BOOST_CHECK_EQUAL_COLLECTIONS(
        result.begin(), result.end(), expected.begin(), expected.end() );
}

void testAdd_nodups() {
    UnderTest inst{}; /* arrange */
    BOOST_REQUIRE_EQUAL(inst.size(), 0u); /* assert */
    inst.add("", ""); /* act */
    BOOST_CHECK_EQUAL(inst.size(), 1u); /* assert */
    inst.add("ENTRY", "entry"); /* act */
    BOOST_CHECK_EQUAL(inst.size(), 2u); /* assert */
    inst.add("OTHER", "other"); /* act */
    BOOST_CHECK_EQUAL(inst.size(), 3u); /* assert */
}

void test_getBestMatch_empty() {
    UnderTest inst{};
    auto result = inst.getBestMatch("any");
    BOOST_CHECK_EQUAL(result, ""s);
}

void test_getBestMatch_one() {
    /* arrange */
    UnderTest inst{};
    inst.add("HOLSDERTEUFEL", "holsderteufel");
    /* act */
    auto result = inst.getBestMatch("ROBERT");
    BOOST_CHECK_EQUAL(result, "holsderteufel"s);
}

void test_getBestMatch_exact() {
    /* arrange */
    UnderTest inst{};
    inst.add("BERLIN", "Berlin");
    inst.add("HAMBURG", "Hamburg");
    inst.add("DORTMUND", "Dortmund");
}
```

```

    inst.add("STUTTGART", "Stuttgart");
    inst.add("WYK", "Wyk");
    /* act and assert */
    BOOST_CHECK_EQUAL(inst.getBestMatch("BERLIN"), "Berlin"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("HAMBURG"), "Hamburg"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("DORTMUND"), "Dortmund"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("STUTTGART"), "Stuttgart"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("WYK"), "Wyk"s);
}

void test_getBestMatch_close() {
    /* arrange */
    UnderTest inst{};
    inst.add("BERLIN", "Berlin");
    inst.add("HAMBURG", "Hamburg");
    inst.add("DORTMUND", "Dortmund");
    inst.add("STUTTGART", "Stuttgart");
    inst.add("WYK", "Wyk");
    /* act and assert */
    BOOST_CHECK_EQUAL(inst.getBestMatch("BRLIN"), "Berlin"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("BURG"), "Hamburg"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("DORTDORT"), "Dortmund"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("STUTGURT"), "Stuttgart"s);
    BOOST_CHECK_EQUAL(inst.getBestMatch("WIK"), "Wyk"s);
}

/* For new test methods: Don't forget to add to init_unit_test_suite() */
};

/* === Suite === */
test_suite* init_unit_test_suite( int argc, char* argv[] ) {
    auto tester = std::make_shared<ImplMultimapTest>();
    auto &ts = framework::master_test_suite();
    ts.add( BOOST_TEST_CASE( [=]() { tester->testConstants(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->testQgramifyEmpty(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->testQgramify1(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->testQgramify2(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->testQgramify3(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->testAdd_nodups(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->test_getBestMatch_empty(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->test_getBestMatch_one(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->test_getBestMatch_exact(); } ) );
    ts.add( BOOST_TEST_CASE( [=]() { tester->test_getBestMatch_close(); } ) );
    return nullptr;
}

```

Listing 14.7 The test suite is manually assembled here.

Now all functions that test the `index_impl` class have become methods of the `ImplMultimapTest` class. `BOOST_AUTO_TEST_CASE` does not help here; instead I define the `init_unit_test_suite` function. Boost.Test usually takes care of this, but this time I do not define `BOOST_TEST_MODULE` and have to write this function myself.

To be able to call the methods, I need an instance of the `ImplMultimapTest` class. This is done by the following line:

```
auto tester = std::make_shared<ImplMultimapTest>();
```

An instance is now available for the duration of the tests. The individual methods must be added to the `framework::master_test_suite()`, which is automatically processed by the Boost.Test framework.

This is how the methods get into this suite:

```
ts.add( BOOST_TEST_CASE( [=]() { tester->METHOD(); } ))
```

You could also use method pointers and `std::bind` here, but I find the lambda clearer.

Note that when you use a `shared_ptr` for the test class, it is *not* recreated for each test case. In this case, it is not critical, but to get the test class reinstantiated for each test case as explained at the beginning, write instead, for example, the following:

```
ts.add(BOOST_TEST_CASE([](){ImplMultimapTest{}.test_getBestMatch_exact();}));  
ts.add(BOOST_TEST_CASE([](){ImplMultimapTest{}.test_getBestMatch_close();}));
```

This makes the constructor and destructor of the `ImplMultimapTest` class somewhat like the *setup* and *teardown* methods of other frameworks, like JUnit. I say this somewhat hesitantly because Boost.Test actually offers a much more flexible handling of *fixtures* for this purpose, such as with the `BOOST_FIXTURE_TEST_CASE` macro. If you are more interested in this, I recommend looking it up in the Boost.Test reference.

14.5.5 Testing Private Members

I already mentioned that it is in the nature of things that unit tests sometimes need to access the private parts of a class. Yes, it is more important to test the visible interfaces, but some complex functions you would rather have tested separately.

Imagine that I want to ensure here that `Q` is really 3. However, `Q` is private, and thus I have no access to it from outside the class.

There are several ways out of this dilemma, but none are ideal. Here, I have written public access methods that start with `_` and are only there to allow access to the fields to be tested within the tests:

```
BOOST_REQUIRE_EQUAL(UnderTest::_prefix().length(), UnderTest::_q()-1);
```

Both PREFIX and Q are private. However, it is important for the functionality of the library that when $Q=3$, the PREFIX has exactly the length 2. Therefore, I take this detour for the comparison via the special access methods.¹⁴

14.5.6 Parameterized Tests

If you want to run a larger set of input data through the same function, you can also have Boost.Test call the same test function with a varying set of parameters.

For example, if you add the code section from [Listing 14.8](#), the testQgramify function will be called sequentially with the elements from the params vector.

```
// https://godbolt.org/z/5GondGsMz (uses boost library)
#include <boost/test/parameterized_test.hpp>
struct Param {
    string input;
    vector<string> expected;
};
const vector<Param> params {
    // { input, expected result }
    {"",      {"^^$"s, "$$"s} },
    {"A"s,    {"^^A"s, "^A$"s, "A$"s} },
    {"AB"s,   {"^^A"s, "^AB"s, "AB$"s, "B$"s} },
    {"ACB"s,  {"^^A"s, "^AC"s, "ACB"s, "CB$"s, "B$"s} },
    {"AAA"s,  {"^^A"s, "^AA"s, "AAA"s, "AA$"s, "A$"s} },
};
void testQgramify(const Param& param) {
    /* arrange */
    UnderTest inst{};
    /* act */
    auto result = inst._qgramify(param.input);
    /* assert */
    BOOST_CHECK_EQUAL_COLLECTIONS(
        param.expected.begin(), param.expected.end(),
        result.begin(), result.end());
}
```

Listing 14.8 A test function with a parameter can be called with test data.

You must then not forget to add the test function to the test suite. To connect Boost.Test data from params with the testQgramify function, use the BOOST_PARAM_TEST_CASE macro:

```
ts.add(BOOST_PARAM_TEST_CASE(&testQgramify, params.begin(), params.end()));
```

¹⁴ For our example, “at least” would still be sufficient, but there are other functions on q-gram indices that require the exact length.

You could also load the data from an external file. Boost.Test can also automatically generate dynamic datasets for you. If you are interested in that, read the Boost.Test documentation, under “Data-Driven Test Cases.”

Don't Forget to Add to the Test Suite!

Actually, I prefer the explicit approach over the automagical one with the `BOOST_AUTO_TEST_CASE` macro.

However, the latter has a huge advantage: when writing a new test at one point in the source code, you can't forget to add it at another point in the code as well. You wouldn't believe how often it has happened to me that I add a new test, which is supposed to fail initially, run the test suite, and then wonder: “Why is it passing?” Quite simply, it's because I forgot to add the new test method to the test suite.

For C++26, there is a proposal for *reflection* (P2996). If implemented, it would be possible for a framework to automatically collect all methods from a test class as test cases. Until then, remember: tie a knot in your handkerchief!

Chapter 15

Inheritance

Chapter Telegram

- **“Has-a” relationship**

Either *aggregation* or *composition*, an object that is part of another or is related to it. A car *has-a* windscreen and *has-a* car port.

- **“Is-a” relationship**

Inheritance; a specialized object is also a more general object. A van *is-a* car.

- **Composition**

Has-a relationship where the object consists of other objects.

- **Aggregation**

Has-a relationship where the object is related to another object.

- **Override**

Redefine a method of a base class in a derived class with the same signature.

- **Virtual method**

A method for which it is decided at runtime which overridden variant is called.

- **Slicing**

When you convert a derived type to a base type and lose information in the process; usually the result of an unintended copy of the wrong type—for example, in call-by-value.

- **Runtime polymorphism**

A base class is declared; at runtime, a derived class is used but retains its derived properties.

Inheritance is a fundamental part of *object-oriented programming* (OOP). Certain techniques and vocabulary accompany this, both of which I introduce in this chapter.

With *inheritance*, you can achieve two things:

- You increase *reusability* and reduce *code duplication*; these are more technical aspects.
- You implement a *design* and thus create clarity in large projects; this is a conceptual aspect.

In this book, I primarily want to teach you the technical aspects and will only briefly touch on the conceptual aspects. However, you should be familiar with them in principle so that you are not at a loss during planning meetings.

15.1 Relationships

Relationships between objects play a central role in OOP as they significantly influence the structure and functionality of software and enable the modeling and organization of complex systems. In this context, the has-a and is-a relationships are of particular importance, as they define how objects are connected and how they relate to each other hierarchically.

15.1.1 Has-a Composition

When you learned about using `struct` and `class` in [Chapter 12](#), I initially bundled data fields into *aggregates*.

```
class Car {
    Windscreen windscreen_;
    std::vector<Wheel> wheels_;
    // ...
};
```

This `Car` *has-a* `windscreen_` as well as `wheels_`—each of the corresponding type. In this case, a has-a relationship, `Car` is even composed of these objects or consists of these objects. This is a *composition*. That means the object usually *owns* its components—which in C++ means that it is responsible for their construction and destruction. The lifetime of the components is limited by the lifetime of the object.



Figure 15.1 Has-a relationships: composition (left) and aggregation (right) in UML notation.

15.1.2 Has-a Aggregation

The situation is different when you say the car *has-a* car port or it *has-a* holder. This is an *aggregation*:

```
class Car {
    CarPort& carPort_;
    Person& holder_;
    // ...
};
```

The contained things are only in a *relationship* with the object, but they do not *belong* to it. Therefore, the object is generally not responsible for their creation or destruction. The lifetime of the elements is independent of the lifetime of the object.

The illustrative example of car, windscreen, and car port seems to clearly describe composition and aggregation respectively. When designing concrete software, things are often not that simple. If you are to program the assembly line of a car manufacturer, you will relate the windscreen instances to the car differently in the database than if you are to create a directory of a car rental company's fleet.

In [Figure 15.2](#), you can see a possible representation of the relationships. Compositions are grouped together in a box. Arrows connect other relationships, aggregations are represented with solid lines and filled arrows. Arrows can also specify the type of relationship for clarity. Within a box, attributes (normal font) and behaviors/methods (*italic font*) are listed.

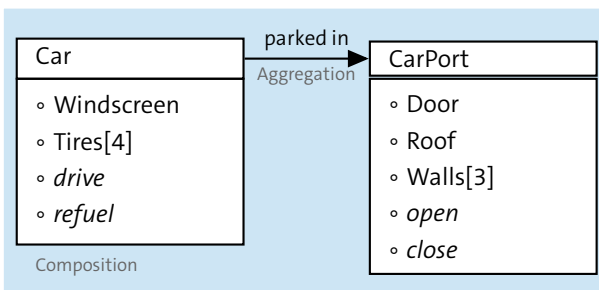


Figure 15.2 An alternative, more informal way to represent the two has-a relationships.

There are dozens of different ways to represent the relationships between objects. I have only given a very brief overview of two of them here.

15.1.3 Is-a Inheritance

With inheritance, you now map an *is-a* relationship. For example: a VW bus *is-a* car. That means the following applies:

- A VW bus has all the properties that a car has.
- Every object that fits the description (or specification) of a VW bus also fits the description of a car.
- The description (or specification) of the VW bus is *more specific*, while that of the car is *weaker*.
- If you expect a car, then it is correct if a VW bus is delivered to you.

With a VW bus and a car, it is obvious that you can map their relationship through inheritance. But sometimes things are not so clear when putting them together. Then these rules might help to check if an *is-a* relationship exists.

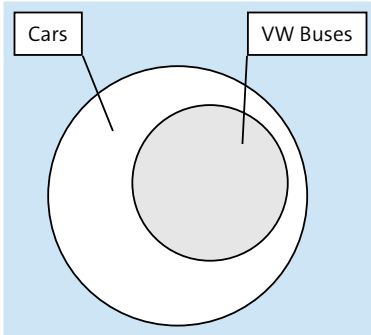


Figure 15.3 This Venn diagram shows that all VW buses are cars, but not all cars are VW buses.

15.1.4 Instance-of versus Is-a Relationship

Note that the is-a relationship for car herbie; does not mean “herbie is-a car.” This is called “is an instance of,” or in C++, “herbie *is of type* car.” It is almost unavoidable that the terms are not always used correctly. For example, “7 is an integer” or “7 is an int” is not precise in this context. But always saying “7 is an element of *integers*” or “7 is of type int” sounds odd.

When you talk about class hierarchies, you should pay attention to this distinction.

Not: Is-an-Instance-of

Remember: In inheritance, *is-a* does not mean an instance of a class, but a subclass that has all the properties of the superclass.

15.2 Inheritance in C++

In C++, you represent inheritance as follows:

```
class Car {                                // Base or superclass
public:
    Windscreen windscreen_;
    std::vector<Wheel> wheels_;
    // ...
};
class VwBus : public Car {                 // VwBulli is a subclass
public:
    Flowers flowers_;
    // ...
};
```

So you write the name of the *base class* separated by a colon : and *public* after the name of the current class:

```
class Subclass : public Baseclass { ...
```

Now you can happily create new *VwBus* instances. And each of them automatically has a *windscreen_* and *wheels_*, even though you didn't explicitly mention it in the *VwBus* class:

```
Vwbus vw{};
cout << vw.windscreen_;
cout << vw.flowers_;
```

If you create a pure *Car* *car*;, then *car.windscreen_* will be a valid access, but in the class *Car*, *car.flowers_* does not exist.

So if you have an instance of the subclass, you can access both its own data fields and methods as well as those inherited from the superclass. For example, if you implement the *camping* method in [Figure 15.4](#), you can use both the *Windscreen* and the *Flowers* in it. The other way around does not work (logically): If you are currently implementing *drive* from *Car*, you can access *Windscreen* and the four *Wheels*, but the *Flowers* do not exist for you.

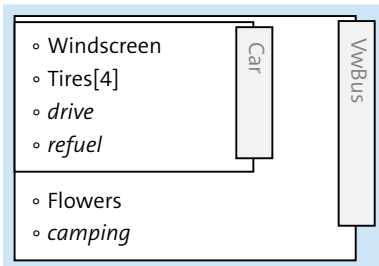


Figure 15.4 All data fields and methods of the superclass are also present in the subclass.

15.3 Has-a versus Is-a

As mentioned earlier, you can design inheritance hierarchies logically and conceptually. How you do this depends on the specific application. Sometimes it makes sense to let the VW bus inherit from *Car*, and in another application, the VW bus is an independent class that has a *Car* as an attribute.

If you are unsure whether to build an inheritance hierarchy, prefer the has-a relationship over the is-a relationship in case of doubt. In C++, inheritance is just a technique to implement the design. In principle, you can achieve the same thing by making *Car* a data element of *VwBus* and passing through the methods for *drive* and *refuel*.

Admittedly, this is quite a strong statement at this point. When you create some classes and hierarchies, you might remember this tip and know that you don't have to squeeze everything into an inheritance hierarchy. Sometimes a well-encapsulated data field will do.¹

15.4 Finding Commonalities

Having said that, I'll steer back to the main point: in situations where you might forgo a seemingly logical inheritance hierarchy, you can, on the other hand, use inheritance to consolidate unrelated data and avoid code duplication. As I said at the beginning, I call this the technical reason for an inheritance hierarchy.

Examine [Chapter 12](#), [Listing 12.24](#) and [Listing 12.27](#), for example. If you write out the data types `Month` and `Day` completely like `Year`, then you have a lot of code that looks identical in three classes. Code duplication is bad—and you can build a small inheritance hierarchy to avoid it.

First, in [Listing 15.1](#), you define the common ancestor `Value` of the three helper classes.

```
// https://godbolt.org/z/hhGo46z5a
#include <iostream> // ostream
#include <format>    // format, vformat, make_format_args
using std::ostream;
class Value {
protected: // not public, only for own and derived use
    int value_;
    const std::string fmt_; // e.g. "{:02}" or "{:04}"
    Value(int v, unsigned w) // constructor with two arguments
        : value_{v}, fmt_{std::format("{{{:0{}}}}", w)} {}
public:
    ostream& print(ostream& os) const;
};
ostream& operator<<(ostream& os, const Value& right) {
    return right.print(os);
}
ostream& Value::print(ostream& os) const {
    return os << std::vformat(fmt_, std::make_format_args(value_));
}
```

Listing 15.1 The common ancestor of our “Year”, “Month”, and “Day” helper classes.

¹ You may lose dynamic type polymorphism, but static type polymorphism with templates is possible.

With this, you have implemented all the important functions of the three value classes. I added the `fmt_` data field because `Year` will be output with a width of four, whereas `Month` and `Day` will be output with a width of two. You add the desired output width as a parameter to the constructor. The `fmt_` format for the `vformat` call in `print` is prepared in the constructor.

Alternatively, you could have written a formatter for `Value` or `Date`, but that would have been more effort.

The declaration of the three helper classes is now very short, as you can see in [Listing 15.2](#).

```
// https://godbolt.org/z/eYY1js4Te
class Year : public Value {           // derive from class Value
public:
    explicit Year(int v) : Value{v, 4} {} // initialize base class
};
class Month : public Value {
public:
    explicit Month(int v) : Value{v, 2} {}
};
struct Day : public Value {           // class-public corresponds to struct
    explicit Day(int v) : Value{v, 2} {}
};
```

Listing 15.2 The duplicate code of the helper classes has now disappeared.

Only the constructors of the respective data types remain; everything else is handled by the inherited class `Value`.

Pay particular attention to the following points:

- In `class Year : public Value`, I am saying with `: public Value` that I am deriving this new class from the `Value` class.
- In the initialization list of the constructors, I call the constructor of `Value` with two arguments; for example, in `Year(int v) : Value{v, 4}...` the base class must also be initialized, and one of its constructors *must* be called. Which one that is, you specify after the colon `:` of the subclass. If you omit this explicit call, the compiler tries to use the base class constructor without arguments. That would have been `Value{}` here, but that does not exist.
- `class Value` begins with a `protected` section. This means that only the class itself and derived classes are allowed to access the content, but not from outside (`public`). The constructor is in this section. This allows the subclasses to call the constructor in their initializations as part of the constructor, but I cannot, for example, define a `Value val{10,3};` in `main`; that would be an access to the constructor “from outside.”

- The base `Value(int v, unsigned w)` constructor does not need to be `explicit` because with two arguments, it is no longer a candidate for automatic type conversion. It wouldn't have been an error, but unnecessary.
- As a reminder: declaring a new type with `class` and immediately starting a `public:` section works the same as if you had defined it with `struct`. I have done this once exemplarily with `struct Day`. What you choose is a matter of taste, but having a guideline is sensible.

The rest of the listing remains mostly unchanged. `Date` uses `Year`, `Month`, and `Day` as usual. For `Date`, the change has remained invisible—an advantage of encapsulation.

```
// https://godbolt.org/z/9rY7qhK89
class Date {
    Year year_;
    Month month_ {1};
    Day day_ {1};
public:
    explicit Date(int y) : year_{y} {} //year-01-01
    Date(Year y, Month m, Day d) : year_{y}, month_{m}, day_{d} {}
    ostream& print(ostream& os) const;
};

ostream& Date::print(ostream& os) const {
    return os << year_ << "-" << month_ << "-" << day_;
}

ostream& operator<<(ostream& os, const Date& right) {
    return right.print(os);
}

int main() {
    Date d1 { Year{2024}, Month{11}, Day{19} };
    std::cout << d1 << "\n"; // Output: 2024-11-19
}
```

Listing 15.3 This is how “Date” uses the new classes.

15.5 Derived Types Extend

In [Chapter 12, Listing 12.24](#), `Easter` was a free function. Wouldn't it make sense if `Easter` were a method of `Year`? Then we could directly ask an instance `Year year{2018}` with `year.easter()` which day Easter falls on in that year.

```
// https://godbolt.org/z/zqKvMedz7 (full example)
class Date; // forward declaration
class Year : public Value {
public:
    explicit Year(int v) : Value{v, 4} {}
    Date easter() const; // declare new method
};
// Declare Month, Day, and Date here. Then:
Date Year::easter() const { // define new method
    const int y = value_;
    int a = value_/100*1483 - value_/400*2225 + 2613;
    int b = (value_%19*3510 + a/25*319)/330%29;
    b = 148 - b - (value_*5/4 + a - b)%7;
    return Date{Year{value_}, Month{b/31}, Day{b%31 + 1}};
}
int main() {
    using std::cout;
    Year year{2025};
    cout << year.easter() << "\n"; // Output: 2025-04-20
}
```

Listing 15.4 Now “easter” is a method of “Year”.

Because I already use `Date` in the declaration of `Year::easter()` in `Date easter() const;` before it is defined, I have to introduce it at the very beginning with `class Date` to indicate that such a type will exist. Before the actual use in the definition of `Year::easter()` from `Date Year::easter() const {...`, the type must then be defined; `class Date { ... }` must precede it.

So far, our three helper classes have had no more functionality than `Value`. Now I have extended `Year` with a method. `Day` and `Month` do not have this method; `Year` is now really different in terms of implementation.

Reuse and extension are both fundamental concepts of OOP.

15.6 Overriding Methods

In the `Value` class, there were no methods in any of the derived classes that had the exact same name or the same *signature*. The signature of a function (or method) is determined by its name and parameter types (including any `const` for this).

In a derived class, you can certainly name a method the same as in the base class. This is called *overriding* the method. This way, you can predefine standard behavior in the base class and redefine it in classes that are “somewhat different” in larger hierarchies.

```

struct Component {
    Color getColor() const { return white; }
};
struct Window : public Component { };
struct MainWindow : public Window { };
struct Dialog : public Window { };
struct TextInput : public Component { };
struct Button : public Component {
    Color getColor() const { return gray; }
};

```

Listing 15.5 All components have a white color; only the button will be gray.

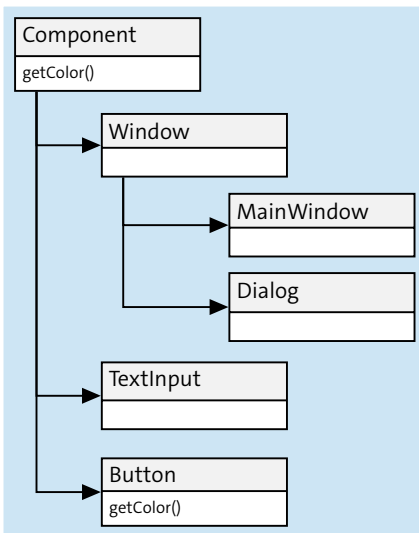


Figure 15.5 A class hierarchy with an overridden method.

If you now write `Component c{}; c.getColor();`, you'll get white just like for `Dialog d{}; d.getColor();` and so on. Only `Button b{}; b.getColor();` returns gray. If you can't deduce the class hierarchy from [Listing 15.5](#), you'll find a visual representation in [Figure 15.5](#).²

15.7 How Methods Work

When using methods in class hierarchies, you need to be aware of a few things. Take a look at [Listing 15.6](#): pay attention to which other methods `print` calls, and consider what you would expect. I have reduced the program to the essentials, which is why it looks somewhat theoretical.

² In practice, such a component hierarchy will not occur this way; the `virtual` keyword, which I will describe later, is still missing.

```

// https://godbolt.org/z/YTK8PfcMc
#include <iostream>
struct Base {
    int eight_ = 8;
    int value() const { return eight_; }
    void print(std::ostream& os) const { os << value() << "\n"; }
};
struct Print : public Base {
    int nine_ = 9;
    void print(std::ostream& os) const { os << value() << "\n"; }
};
struct Value : public Base {
    int ten_ = 10;
    int value() const { return ten_; }
};
struct Both : public Base {
    int eleven_ = 11;
    int value() const { return eleven_; }
    void print(std::ostream& os) const { os << value() << "\n"; }
};
int main() {
    Base ba{}; ba.print(std::cout); // Base call
    Print pr{}; pr.print(std::cout); // print overridden
    Value va{}; va.print(std::cout); // print from Base
    Both bo{}; bo.print(std::cout); // everything overridden
}

```

Listing 15.6 What does “print” output? The “value” method appears more often.

What do the different print lines in main output in your opinion?

■ **Base ba{}; ba.print(cout);—base call**

In Base, the two print() and value() methods are defined directly. No other class is involved here yet. value() thus returns eight_, and 8 is output.

■ **Print pr{}; pr.print(cout);—print overridden**

In Print, the print() method is *overridden*. For pr, the Print::print() method is called. value() is needed there. This method was inherited from Base. And Base::value() returns eight_—so 8 is output.

■ **Value va{}; va.print(cout)—print from Base**

we.print() must fall back on the inherited method Base::print(). Now the value() method is needed there. Although there is the Value::value() method, it is *not yet known* in Base::print()! In a method of the base class, only methods of the base class are used—here, Base::value(), which returns eight_ and leads to the output of 8.

■ **Both `bo{}`; `bo.print(cout)`;—everything overwritten**

Here it is simple again: There is a `Both::print()` that gets called. The call to `value()` accesses `Both::value()` in its own method and refers to `eleven_`. It outputs 11.

Did you correctly guess the third case? If not, don't be upset. Which `value()` method is taken by `Base::print()` is a matter of definition: both are possible. In C++, it is defined such that a method only sees the methods available at the time of the class's translation through inheritance, new definition, or overriding. `Base::print()` knows nothing about potentially derived classes yet.

15.8 Virtual Methods

In other languages, this is sometimes different: if you had written the preceding listing (with obvious changes for the language) in Java, then you would have seen a 10 at `Value va{}`; `va.print(cout)` as Java checks at *runtime* which methods are available to an instance.

And because this is also a reasonable behavior (otherwise it would hardly have been chosen as the default behavior in Java), it is also possible in C++. The key to this is *virtual methods*.

If you mark a method with the `virtual` keyword, the compiler decides at *runtime* which version of this method is valid. And thus you get the Java behavior.

```
// https://godbolt.org/z/vW5s3Kh6M
#include <iostream>

using std::ostream; using std::cout;
struct Base2 {
    int eight_ = 8;
    virtual int value() const           // virtual method
    { return eight_; }
    void print(ostream& os) const
    { os << value() << "\n"; }
};
struct Value2 : public Base2 {
    int ten_ = 10;
    virtual int value() const override // override
    { return ten_; }
};
int main() {
    Value2 v2{}; v2.print(cout);      // use
}
```

Listing 15.7 Methods marked with “virtual” are resolved at runtime.

`Value2 v2{}; v2.print(cout);` corresponds to the `Value v{}; v.print(cout);` call from the previous listing. Now you will see that `10.v2.print()` must refer to the `Base2::print` definition because `Value2` has not defined this method itself, but the call to `value()` in `print` is now a *virtual method call*: it is decided at runtime. And because `v2` is of type `Value2`, `Value2::value` is used, `ten_` is returned, and thus `10` is output.

Virtual and Nonvirtual

Virtual method calls are decided at runtime, normal method calls at compile time.

In [Listing 15.7](#), in the definition of the `value()` method, you see the `override` keyword in addition to the `virtual` keyword. Such an additional designation requires the compiler to ensure that this method indeed overrides another method. This way, you can avoid producing hard-to-find errors due to a typo, for example. Imagine, for instance, that you accidentally named the method `virtual int value() const` in `Value2`. The program will compile and run, but it will not return the values you expect.

Or consider this other, quite practical example. In a `mydefs.hpp` header file, some types are defined, such as `using value_t = int;`. And now you have a small hierarchy:

```
struct Number {
    virtual void add(value_t value);
};

struct SafeNumber : public Number {
    virtual void add(int value);    // override not specified, int instead of value_t
};
```

The fact that in `Number::add()` the parameter is of type `value_t`, but in `SafeNumber::add()` it is specified as type `int`, is the same for the compiler. `using` (and the corresponding typedef) is just a type alias, not a completely new type. Therefore, `SafeNumber::add()` overrides the predecessor as desired.

If you now change the definition in `mydefs.hpp` to `using value_t = long;`, then the signature of `Number::add()` changes. However, the signature of `SafeNumber::add()` does not change because you did not use the type alias. You no longer override the original method and will very likely get undesired behavior in your program—in the form of a hard-to-find bug. If you had marked `SafeNumber::add()` with `override`, then the compiler would have pointed out the error.

Use “override”

When you override a virtual method, also specify `override`. This way, you can avoid hard-to-find errors.

15.9 Constructors in Class Hierarchies

Constructors are special class elements: they are *not* methods, and therefore they are not inherited to derived classes like methods.

// <https://godbolt.org/z/59WKfxenE>

```
class Base {
public:
    Base() {} // null-argument constructor
    explicit Base(int i) {} // one argument
    Base(int i, int j) {} // two arguments
    void func() {}; // method
};

class Derived : public Base { // no own constructor
};

int main() {
    Base b0{}; // okay, null-argument constructor
    Base b1{12}; // okay, one argument
    Base b2{6,18}; // okay, two arguments
    Derived d0{}; // okay, compiler generates default constructor
    d0.func(); // okay, method is inherited
    Derived d1{7}; // ✗ Error: no constructor for one argument
    Derived d2{3,13}; // ✗ Error: no constructor for two arguments
}
```

Listing 15.8 The derived class inherits methods but not constructors from the parent class.

The attempt to create a new object with `Derived d1{7}` fails because `Derived` does not define its own constructor for an `int`. The `Base(int)` constructor is not inherited, as is the case with normal methods—for example, with `func()`, which you can also call for `Derived` instances.

The reason for this is that constructors are really different from methods. A constructor must initialize a class instance. This is a task that is closely tied to the internal structure of the class. Calling a parent class constructor instead of its own constructor is very likely not achieving a correct initialization of the derived class. There may be administrative information related to the class that needs to be initialized in the respective constructor. What administrative information is needed is not specified in detail by the standard; it is left to the implementation. To allow this freedom, constructors have this special role.

If you still want to inherit the parent class constructors, you must explicitly specify this in the derived class. To achieve this, include `using Base::Base;` in the class definition,

thereby explicitly referencing the constructors. Yes, constructors, plural—because this approach elevates all inherited constructors simultaneously, rather than just one. You can only employ this mechanism by following the all or nothing principle.

```
// https://godbolt.org/z/h15eGahjh
class Base {
public:
    Base() {}
    explicit Base(int i) {}
    Base(int i, int j) {}
    void func() {};           // method
};

class Derived : public Base {
public:
    using Base::Base;         // importing all parent class constructors
};

int main() {
    Derived d0{};             // okay, imported, no longer generated
    Derived d1{7};            // okay, was imported
    Derived d2{3,13};         // okay, was imported
}
```

Listing 15.9 Using “using” to import all parent class constructors.

In this way, you elevate all parent class constructors to your own class, and the necessary additional management tasks are handled by the compiler.

15.10 Type Conversion in Class Hierarchies

Let's stick with our abstract example, but let me expand it a bit with dangerous code.

15.10.1 Converting Up the Inheritance Hierarchy

Consider the following listing.

```
// https://godbolt.org/z/Ec7vqEdY3
//... Base2 and Value2 as before ...
void output(Base2 x) {       // pass by value
    x.print(cout);
}
```

```
int main() {
    Base2 ba2{}; output(ba2);    // outputs 8
    Value2 va2{}; output(va2);  // also outputs 8
}
```

Listing 15.10 Passing by value only copies the common part of the type.

By passing by value, `va2` is copied into the parameter `x`. Because the parameter is of type `Base2`, only the part of `va2` that belongs to `Base2` is copied. And because `x` is now of this type, `x.print()` can only do what any other variable of this type would do in `Base2::print()`—that is, output 8.

15.10.2 Downcasting the Inheritance Hierarchy

If you had chosen the derived type for the parameter instead of the base type—that is, `output(Value2 x)`—then `output(va2)` would have output 10. However, `ba2` does not match the `Value2` parameter type, so the compiler responds with an error when converting in this direction of the hierarchy. Instances of the base type do not have all the properties needed to be converted into the derived type; the compiler cannot just make up properties to fill in.

```
// https://godbolt.org/z/17nah59eP
//... Base2 and Value2 as before ...
void output(Value2 x) {          // derived class as value
    x.print(cout);
}
int main() {
    Base2 ba2{}; output(ba2);    // ✗ ba2 cannot be converted to Value2
    Value2 va2{}; output(va2);  // outputs 10
}
```

Listing 15.11 The derived class as an argument type to a function does not allow calling with a base class variable as a value parameter.

15.10.3 References Also Retain Type Information

When you pass an instance as a reference, it does not need to be copied. In this case, the object remains as it is—along with its actual type.

```
// https://godbolt.org/z/sW7MexWrh
//... Base2 and Value2 as before ...
void output(Base2& x) {          // passing as a reference
    x.print(cout);
}
```

```
int main() {
    Base2 ba2{}; output(ba2); // outputs 8
    Value2 va2{}; output(va2); // outputs 10, because the object is not copied
}
```

Listing 15.12 Passing by reference does not change the instance.

When `va2` becomes `x`, it is only “renamed.” `x` remains essentially a `Value2`—and thus `print` can access the virtual `Value2::value` method and output 10.

Runtime polymorphism is in play when a more general class is used in a declaration, but at runtime, a more specific class is used for the declared variable without losing its properties. In the example, although the parameter is declared as `Base2`, `output` can be called at runtime with a `va2` instance of the `Value2` class, *and* `va2` retains its property of outputting 10—which it would not do if it were completely converted to the `Base2` type. You achieve this form of polymorphism in C++ only when you work with references (or pointers).

In case you’re wondering, if `Base2::value()` were not virtual (just like `Base::value()`), you would be back to the behavior from [Listing 15.6](#) and would receive an 8.

15.11 When to Use Virtual?

Whether you mark a method with `virtual` or not depends on the *design* you choose. In C++, it is not common practice to make all methods of all classes virtual by default. There can be reasons for and against making each individual method virtual, and the same applies to each individual class. This is somewhat different from the reasons for choosing between methods and free functions: encapsulation with methods is better, and so you usually opt for them. When it comes to deciding whether to use virtual methods or not, the matter is not quite as clear. Take the following hints as decision aids for your design:

- Some classes primarily serve the purpose of holding data together. They are not part of a hierarchy. Without inheritance, there is no reason for virtual methods.
- The existence of a class hierarchy alone does not justify virtual methods. Maybe you are just deduplicating very skillfully.
- Overriding a method within a class hierarchy is a good candidate for a virtual method, but not necessarily.
- If your design relies on overridden methods—that is, changing behavior from class to class—then `virtual` is appropriate.
- Constructors are never virtual. You must not call virtual methods within constructors.

- A destructor (see [Chapter 16](#)) must be virtual if there is at least one virtual method in the class.
- A virtual method of a base class that you override is automatically also virtual, even if you don't explicitly say so. Once `virtual`, always `virtual`.

Classes Either Entirely without or Entirely with “virtual”

For starters, I recommend following these two rules:

- Without a hierarchy, make nothing `virtual`.
- With a hierarchy, make *all* methods `virtual` if you override at least one method.

If you keep this in mind, you'll likely design your classes (and hierarchies) in a way that naturally leads to the decision of whether a method should be `virtual` or not.

But why ask the question at all? What are the advantages and disadvantages of virtual methods?

The big advantage is that you gain a more flexible design. You can modify behavior in derived classes even if it was originally defined in a base class.

The disadvantages concern speed and memory:

- **The method must first be looked up in a table at runtime.**

When you call a virtual method, this call requires one more indirection (and possibly an addition). But modern processors handle this effortlessly, and you will not notice any speed differences compared to a normal method call.

- **Virtual methods can rarely be optimized into inline functions.**

You might notice this more in the program's speed, especially if you call a virtual function within a tight loop. Inlining is a crucial compiler optimization, and its importance increases with the complexity and modernity of the processor executing the program.

- **A hidden data field is required per instance.**

Each instance of a class with at least one virtual method has an additional hidden variable. This is implemented in most C++ compilers: it is a pointer (`vptr`) to a static table (`vtable`). If you have small instances that you pack in large quantities—for example, in a container—it will affect memory usage.

You could also argue that you want to align your design with other current programming languages. Java, for example, does not have any nonvirtual methods. I advocate at least separating *data-holding classes* (*data transfer objects* [DTOs]) from *behavior-oriented classes* (with nontrivial methods) in the design. DTOs do not need virtuality, but business logic classes might, depending on the purpose. Someone who decides to make all methods virtual “just to be sure” should consider whether this approach is suitable for their application. While I can understand the reasoning behind this design decision, it's essential to ensure that it is appropriate.

15.12 Other Designs for Extensibility

If you decide not to use virtual methods, there are other ways in C++ to make data types extensible. You can add functionality by overloading free functions. Linking types using *type traits* is an advanced topic. For this, you need to learn more about template programming. You can find a brief introduction in [Chapter 23](#).

Feel free to use virtual methods generously. They offer a good compromise between understandability and performance. If you ever deal with special applications, you can always learn other methods.

A general note on inheritance: Practice the technique of inheritance and use it to reduce code duplication. But don't overuse the concept. Much more often than the *is-a* relationship, inheritance, is the *has-a* relationship, composition or aggregation, suitable for meaningfully connecting classes and objects. Don't force a bunch of classes into the rigid framework of an inheritance hierarchy come hell or high water. In practice, designs turn out to be more flexible when they can be extended and modified more through data fields than through a class hierarchy.

While you can model composition in C++ with the overloading of free functions or type traits, interfaces are known from languages like Java. The equivalents in C++ are abstract classes with only pure virtual methods; see [Chapter 16](#). This technique is also common in C++, but it is not used as widely as in Java.

Chapter 16

The Lifecycle of Classes

Chapter Telegram

- **Destructor** `Type::~Type()`

The opposite of the constructor; called when an object is removed.

- **Resource acquisition is initialization (RAII)**

Programming technique; enabled by classes that request resources in the constructor and release them in the destructor.

- **Temporary value** (`temp-value`)

A value without a variable name, usually within an expression or part of a type conversion as a function argument; removed at the end of the statement.

- **Temp-value reference**

A reference to a temporary value; symbol `&&`. Technically *rvalue reference*, as it stands for expressions that can only appear on the right-hand side of an assignment.

- **operator@**

Where `@` stands for any valid operator symbol. The free function or method you need to define so that your type supports the operator.

- **Type::Type(const Type&)**

Copy constructor; called by the compiler when a new instance is to be created from an existing one, including function returns and parameter passing by value.

- **Type& Type::operator=(const Type&)**

Assignment operator. The compiler calls the assignment operator when a new value is assigned to an already existing instance.

- **Move operation**

A special form of copying from a temp-value. Programmers can optionally specify that object contents are transferred instead of copied. The standard library consistently supports this form.

- **Type::Type(Type&&)**

Move constructor; for creating a new instance from a temp-value as the source.

- **Type& Type::operator=(Type&&)**

Move assignment operator; assigning the content of a temp-value to an existing instance.

- **= delete**

The explicit deletion of functions, methods, constructors, and the like.

- **enum class**
Enumeration type; a type whose instances can take on a countable number of possible values. Each possible value gets its own identifier.
- **friend**
Friend functions are free functions that are granted `public` access rights to a class. `friend` is often used for the implementation of operators.
- **Abstract method (pure virtual)**
Virtual method that you define with `= 0` instead of an implementation.
- **Abstract class**
Class with at least one abstract method; cannot be instantiated itself, but must be derived.

So far, you have managed everything by initializing classes with a constructor, providing them with data fields, and accessing them with methods. When dealing with instances of these classes, many things happen in the background that you should be aware of. On the one hand, you can use these automatisms and incorporate them into the design to save yourself work, but on the other hand, they can also become a stumbling block if you are not familiar with them.

16.1 Creation and Destruction

A class instance begins its life the moment you *define* it. At that moment, its constructor is called. You can use it until the program leaves its *scope*; as a rule of thumb, this is usually the closing curly brace `}` of the current code section. You have already seen this for the built-in types in [Chapter 8, Listing 8.4](#), but it also applies to classes. When the last statement before this brace is executed, the instance is removed, and this has the following main consequences:

- In the program code—that is, at compile time—you can no longer access the variable.
- At runtime, the instance is removed from memory, and its *destructor* is called.
- If there is still a *reference* `&` to this instance from another place, you must not use it anymore. If you do, it is an error and will—at best—lead to a program crash.

To clarify these important points once again, I have illustrated various possibilities in [Listing 16.1](#) for when an object is created and when it is destroyed (function stands for arbitrary functions).

```
// https://godbolt.org/z/9jGKPbzqo
struct MyValue { /* something */ };
MyValue globalValue{};                               // global class instance
```

```

void function(const MyValue &paramRef) {
    if( /*...*/ ) function( /*x1?*/ );    // call some function
    MyValue localValue{};                // local class instance
}                                          // end of function

int main() {
    MyValue mvalue1{};
    function( /*x2?*/ );
    function( MyValue{} );                // temporary value
    {
        MyValue mvalue2{};
        function( /*x3?*/ );
        MyValue mvalue3{};
    }                                     // end of inner block
    function( /*x4?*/ );
    MyValue mvalue4{};
    function( /*x5?*/ );
}                                          // end of main function

```

Listing 16.1 Which variables can you use for “x1” to “x5”?

The *validity* is the simpler topic. Can you see which of the variables you can use in the various function calls of `function( /*...?*/ )`? An overview is shown in [Table 16.1](#).

Variable	x1	x2	x3	x4	x5
globalValue	Yes	Yes	Yes	Yes	Yes
paramRef	Yes				
localValue					
mvalue1		Yes	Yes	Yes	Yes
mvalue2			Yes		
mvalue3					
mvalue4					Yes

Table 16.1 When can what variables be used in “function()”?

Closely related to validity is the question of when something is *created* and when it is *destroyed*. The life of each variable begins with its *definition*—that is, when it is *initialized*. In the case of classes, you can also say that they are *constructed* because a *constructor* is called.

The life of a variable ends when its scope is exited. Specifically, in our example, this means the following:

- The `globalValue` variable is destroyed upon exiting `main`.
- The `paramRef` variable is destroyed upon exiting `function`.
- The `localValue` variable is destroyed upon exiting `function`.
- The `mvalue1` variable is destroyed upon exiting `main`.
- The `mvalue2` variable is destroyed upon exiting the inner block.
- The `mvalue3` variable is destroyed upon exiting the inner block.
- The `mvalue4` variable is destroyed upon exiting `main`.
- The `MyValue{}` variable is destroyed at the ; of the line.

16.2 Temporary: Short-Lived Values

The line with the temporary value is particularly noteworthy. With `MyValue{}`, you create an instance of the class, which is then passed as a parameter to `function`. However, the instance does not get a variable name. Instead, you create a *temporary value* (also called a *temp-value*). As such, this value disappears at the end of the statement—as a rule of thumb, at the next semicolon.

When you create a temp-value, it is not a *declaration*, but an *expression*. Therefore, you can use it directly—for example, as a parameter for a function.

There are other ways to generate temp-values. For example, an automatic type conversion can lead to this, as shown in the following listing when calling `length()`.

```
// https://godbolt.org/z/eK1Ervjs5
#include <string>
#include <iostream>                                // cout
using std::string; using std::cout;

struct Value {
    int value_;
    Value(int value)                                // 1-arg constructor = type conversion
        : value_{value} {}
};

size_t length(string arg) {
    return arg.size();
}
Value twice(Value v) {
    return Value{ v.value_*2 };
}
```

```
int main() {
    cout << length("Hipp hopp") << "\n"; // const char* to string
    cout << twice(10).value_ << "\n";    // int to Value
    string name {"Gandalf"};
    cout << ( name + " the Grey" ) << "\n"; // string + const char*
}
```

Listing 16.2 When you call a function with a parameter for which the compiler invokes a constructor to perform a conversion, it creates a temp-value.

The `length` function takes a string as an argument. However, when I call `length("Hipp hopp")`, I am passing a text literal, which is a `const char[]`. The compiler automatically creates a string just for the purpose of passing it to `length()`. After the statement, this created string disappears because it is a temp-value.

The mechanism the compiler uses is already known to you: automatic type conversion. The same thing happens with `twice(10)`. Because `Value` has a constructor with one argument, the compiler can use it for automatic type conversion. And lo and behold, `10` is an `int` and fits—so a temporary `Value` is created, used within `double` as `v`, and discarded at the end.

Especially binary operators like to create temporary values, such as `plus +`. The `name + " the Gray"` expression is the call of the `operator+` with the `string` and `const char*` argument types. The operator concatenates the arguments into a new string, which is then output. The concatenated result is the temp-value and is no longer needed after the `;`.

Background: Lvalues, Rvalues, and Legal Theft

In the context of temp-values, you will often encounter terms like *lvalue*, *rvalue*, and *rvalue references* in reference books and on the internet. These are not features that C++ brings, but rather the technical terms used to describe how C++ implements certain concepts. As a concept behind *rvalue*, I find the term *temp-value* better. Nevertheless, when trying to understand the how and why, you will often come across the technical terms. In the following list, assume you have declared `int x`, `*p`, `a[10]`:

■ Lvalue

An *expression* to which you can assign something, like `x=1`, `*p=1` or `a[2]=1`. The letter *l* comes from *left*, because these can appear on the left side of an assignment—that is, `lvalue = ...`.

■ Rvalue

An *expression* to which you cannot assign anything, like `42=1`, `&x=1` or `x+1=1`. These are expressions that can *only* appear on the right side of an assignment—that is, `... = rvalue`.

The distinction is important because an rvalue only exists briefly (if at all)—hence my term, *temp-value*. The content of temp-values can be changed without affecting any other part of the program. Typically, one transfers a resource from a temp-value to another object to save a copy. This is important for understanding move operations, which are discussed in [Section 16.9](#).

16.3 The Destructor to the Constructor

If you want to check when an object is destroyed, then do it in the *destructor*. As a counterpart to the constructor, it is used to perform cleanup tasks. This can involve clearing the class's data fields or other administrative tasks.

Because you are only familiar with data structures that are automatically deinitialized, output a text as an administrative task. Fill the `MyValue` class from the introductory example with some life.

```
// https://godbolt.org/z/WPcTv3zKM
#include <string>
#include <iostream>
#include <iomanip> // setw
using std::cout; using std::setw; using std::string;
struct MyValue {
    static int counter;           // static: exists only once for all instances
    int number_;                 // indentation level of this instance for output
    string name_;                // name of this instance for output
    explicit MyValue(string name)
        : number_{++counter}     // increment indentation level per instance
        , name_{name}           // remember the object's name for output
    {
        cout << setw(number_) << " " // use number_ for indentation
              << "Constructor " << name_ << "\n"; // output instance name
    }
    ~MyValue() {                 // destructor
        cout << setw(number_) << " " << "Destructor " << name_ << "\n";
    }
};
int MyValue::counter = 0;       // initialization of the static class variable
```

Listing 16.3 A destructor is executed when an object is removed.

Equipped like this, you can use the object as usual. To do this, use specific variables of this modified `MyValue` type in [Listing 16.1](#).

```
// https://godbolt.org/z/ennEoWPzb
void function(const MyValue &paramRef) {
    MyValue localValue{"local"};
}

int main() {
    MyValue mvalue1{"mvalue1"};
    function( MyValue{"temp"} );
    function( mvalue1 );
    {
        MyValue mvalue2{"mvalue2"};
    }
}
```

Listing 16.4 Here, many objects are created and destroyed.

Thus, you get the output shown in the following listing.

```
Constructor mvalue1
Constructor temp
Constructor local
Destructor local
Destructor temp
Constructor local
Destructor local
Constructor mvalue2
Destructor mvalue2
Destructor mvalue1
```

Listing 16.5 This output shows when objects are created and destroyed.

In `main`, `mvalue1` is created first. You used `setw` to ensure that the first instance of `MyValue` is indented to a depth of 1, and with `++counter`, all subsequent instances are indented one level deeper. The counter value is simultaneously copied into the `number_` data field. This way, you can use the same indentation level in the destructor as in the constructor. You can clearly see that `mvalue1` is also the last object to be cleaned up: the `Destructor mvalue1` output occurs in the destructor.

You also see that `temp` is created for the function call. Within the function, `local` is created and removed again—for each of the two calls to `function`. After returning from the function, `temp` is removed—at the end of the statement—while `mvalue1` remains.

Next, `mvalue2` is created. However, the end of the inner block is immediately reached, which is why its destructor is called right after. Only then is the end of `main` and thus the end of the validity of `mvalue1` reached.

16.3.1 No Destructor Needed

Normally, it is not necessary to do anything in the destructor because every data field of a class is cleaned up anyway. So if you do nothing, leave the destructor empty, or omit it entirely, all data fields will still be removed.

Just use the mechanisms we've covered so far and you won't need a destructor for cleanup. Basic data types like `int` and `double`, along with standard library types like `string`, `vector`, and `fstream`, manage everything automatically.

16.3.2 Resources in the Destructor

But there are resources that are not automatically removed. Raw pointers and many C data types must be handled in the destructor. Alternatively, you can wrap the whole thing in a `shared_ptr` and use a *custom deleter*. You will learn more about this in [Chapter 20](#).

Suppose you are dealing with a database that is provided to you via a library and with the associated header as a programming interface. Simplified, the header might look like the following listing.

```
// https://godbolt.org/z/54dzb31z
#ifdef DATABASE_HPP
#define DATABASE_HPP

typedef void* db_handle_t;

db_handle_t db_open(const char* filename);
void db_close(db_handle_t db);
int db_execute(db_handle_t db, const char* query);

#endif
```

Listing 16.6 A simple example of a C interface to a resource.

Thus, it is clear that for every `db_open()`, you must also call a `db_close()`. If you don't, all sorts of things can happen: writes to the database might be forgotten, your program might leak database connections and eventually be unable to perform any more `db_open()`s, or your program might crash.

To ensure that for every `db_open()` a `db_close()` is called, get help from the compiler: constructor calls and destructor calls are always paired. Wrap the resource in a *resource wrapper*.

```
// https://godbolt.org/z/6399qK9Ts
#include <iostream>           // cout
#include "database.hpp"       // include the foreign API
```

```

class Database {
    db_handle_t db_;           // wrapped resource
public:
    Database(const char* filename);
    ~Database();
    int execute(const char* query);
};
Database::Database(const char* filename)
    : db_{db_open(filename)}    // requesting the resource
    { }
Database::~~Database() {
    db_close(db_);             // releasing the resource
}
int Database::execute(const char* query) {
    return db_execute(db_, query); // using the resource
}
int main() {
    Database db{"customers.dat"}; // creating the wrapper
    std::cout << "Count: " << db.execute("select * from cust") << "\n";
}
// automatically removing the wrapper

```

Listing 16.7 If you need to close a resource, the destructor is suitable for this.

In the constructor, you request the resource—the database connection—with `db_{db_open(filename)}` and store it in the private `db_` data field. You release this resource in the destructor with `db_close(db_)`. But do continue reading this chapter as you are still missing the copy constructor and the assignment operator.

You have omitted the necessary error checks here, which you must of course include in a real program. With such C-like interfaces, it is common that certain return values mean something special. For example, the return of `db_open` could have been checked to react specifically in case of an error.

```

Database::Database(const char* filename)
    : db_{ db_open(filename) }
    {
        if(nullptr == db_) { // Error opening
            throw IllegalArgumentException("Error opening the DB");
        }
    }
}

```

Listing 16.8 The constructor initializes or throws an exception.

It is important here that the constructor exits with an exception in case of an error, and not in the normal way. This means the object is considered not created. Where no object was created, none needs to be removed. As a result, no destructor is called.

An Exception in the Constructor Means No Destructor Call

If an exception is thrown and not caught inside the constructor, the object is considered not created. A destructor is then not called.

This is important here because if `db_` is `nullptr` after the initialization error, then the call to `db_close(db_)` could become problematic. However, by exiting the constructor with an exception, the destructor is not called, and thus no problem arises.

You will learn how to handle exceptions in the next section. Here, you have learned how to use destructors effectively for *resource acquisition is initialization* (RAII), meaning that when you request a resource, you do so through initialization—in this case, by initializing the `Database` wrapper class.

Be sure to include RAII in your repertoire as it helps you avoid hard-to-find errors. Remember that I mentioned a few sections earlier ([Section 16.3.1](#)) that you usually don't need a destructor when dealing with standard library types? Well, the reason for that is that those types have implemented this concept consistently. You rarely need to make paired acquire and release calls. This is carefully packaged in the constructors and destructors of the classes.

16.4 Yoda Condition

In the condition of the `if` statement in [Listing 16.8](#), I have hidden a little treat:

```
if(nullptr == db_) ...
```

I compare `nullptr` and `db_` with `==`. I could just as well have written `db_ == nullptr` and achieved the same result. However, the order used has a big advantage.¹ If you accidentally type `=` instead of `==`—which is something entirely different—then the compiler will point out the error: because `nullptr` is a literal, you cannot assign anything to it with `=`. If you had accidentally written `db_ = nullptr`, you would have unintentionally written an assignment. The assignment is also an expression, whose result here would then be `nullptr`—and `nullptr` is automatically converted to `false` by the compiler.

So if you implement a comparison with a literal or a constant, you might get into the habit of writing the constant on the left side of the comparison for your own safety. You then have slightly greater protection against typos.

This form of writing a condition is called a *Yoda condition* because you write the operands in reverse order compared to what you would naturally do. Or, as Yoda might say: naturally do it, you would. Hence the name.

¹ At least so long as the order of the operands does not matter to the comparison, as in this case.

The following listing offers some other examples of Yoda conditions in which you are protected against accidental `=`.

```
if("Yoda" == character) ...
if(42 == answer) ...
```

Listing 16.9 Examples of Yoda conditions with `"=="`.

Calling methods in this pattern also falls under the Yoda conditions umbrella.

```
#include "my_string.hpp"
static const my_string ZEBRA { "zebra" };
int main() {
    my_string animal{ "horse" };
    if(ZEBRA.equals(animal)) return 0;
    else return 1;
}
```

Listing 16.10 A Yoda condition with a method call.

Here you call the `equals` method on the constant, instead of putting the variable in front in `animal.equals(ZEBRA)`. You find this notation less often in C++ than, for example, in Java. There, `animal` could be an invalid object, something similar to `nullptr`, and that would definitely be an error for a method call with `animal.equals(ZEBRA)`. In the call to `ZEBRA.equals(animal)`, since `ZEBRA` cannot be the `nullptr`, you have the chance to safely check within the `equals` method if its parameter (here, if `animal`) is the `nullptr`, and handle that scenario accordingly.

The `std::string` class does not have an `equals` method like in Java. You compare using `==`. With `==`, the Yoda notation does not significantly affect readability. However, many believe that readability suffers with methods, so consider whether you want to adopt this notation.

16.5 Construction, Destruction, and Exceptions

The RAII concept also includes that the object is considered not created if the initialization (constructor run) fails (causes an exception). This has the consequence that no destructor is called either.

```
// https://godbolt.org/z/99PPn595E
#include <iostream> // cout
#include <stdexcept> // runtime_error
struct CanThrow {
    CanThrow(int whatShouldHappen) {
        std::cout << "Constructor " << whatShouldHappen << "...\\n";
```

```
        if(whatShouldHappen == 666)
            throw std::runtime_error("Test error");
        std::cout << "...Constructor finished\n";
    }
    ~CanThrow() {
        std::cout << "Destructor.\n";
    }
};

int main() {
    try {
        CanThrow ct1{0}; // okay, does not throw an exception
    } catch(std::runtime_error &exc) {
        std::cout << "Caught-1: " << exc.what() << "\n";
    }
    try {
        CanThrow ct2{666}; // throws, ct2 is not created
    } catch(std::runtime_error &exc) {
        std::cout << "Caught-2: " << exc.what() << "\n";
    }
}
```

Listing 16.11 The constructor of “CanThrow” can terminate with an exception.

This program will provide you with the following output:

```
Constructor 0...
...Constructor finished
Destructor
Constructor 666...
Caught-2: Test error
```

That means that for `ct1`, the constructor was fully executed. The object was fully created, and `ct1` is available to the program until the end of the block.

For `ct2`, the constructor was started but exited with an exception. Thus, `ct2` is considered not created. Therefore, you do not see a call to the destructor of `ct2`.

If your class does more than `CanThrow`, such as initializing some data fields, then all objects created up to the exception will be cleaned up as if their scope was exited (so do not use `new` together with raw pointers here).

```
class Mega {
    std::vector<int>    data_;
    CanThrow           canThrow_;
    std::map<string,int> more_;
```

```
public:
    Mega()
        : data_{}
        , canThrow_{666} // triggers an exception
        , more_{}
        { }
};
```

Listing 16.12 Partially initialized data fields are also cleaned up in the case of an exception.

An exception occurs during the initialization of `canThrow_`. At that point, `data_` has already been created. No problem: the exception leaves the constructor, but `data_` is properly cleaned up beforehand. `more_` has not been initialized yet.

A Destructor Must Not Throw an Exception

Because during the handling of an exception, other objects may need to be cleaned up using a destructor—as shown with `data_` in [Listing 16.12](#)—you should never let an exception escape from a destructor. You can assume that no destructor in the standard library throws an exception if your types do not either.

The title of this box simplifies the matter a bit, but it is useful for keeping the point in mind. If another exception is thrown while processing an exception, your program will terminate immediately.

16.6 Copy

Let's make a small change to the definition of `function` in [Listing 16.13](#): pass the parameter by value instead of as a constant reference. As you know, this will copy the object for the function. The rest of the program remains the same.

```
// https://godbolt.org/z/3T869arc8
void function(MyValue paramValue) {
    std::cout << "(function)\n";
    MyValue localValue{"local"};
}
int main() {
    MyValue mvalue1{"mvalue1"};
    function( MyValue{"temp"} );
    function( mvalue1 );
    {
        MyValue mvalue2{"mvalue2"};
    }
}
```

Listing 16.13 Passing by value creates objects with the compiler-generated copy constructor.

I added the output of (function) just for orientation; it has nothing to do with the lifetime of the instances.

There is something strange to see in the output:

```
Constructor mvalue1
Constructor temp
(function)
Constructor local
Destructor local
Destructor temp
(function)
Constructor local
Destructor local
Destructor mvalue1
Constructor mvalue2
Destructor mvalue2
```

Destructor mvalue1

You see with Destructor mvalue1 that mvalue1 is cleaned up twice. This is the result of negligence in our code. We have already mentioned this: when passing a parameter by value, the object is *copied*. So you get a second object, and as such, it must of course also be cleaned up again—including the corresponding destructor call.

The object in question was not created by calling our constructor; you would have seen that. No, for the purpose of copy there is a special constructor, the copy constructor. You can define it yourself—and you should in this case, because only then will our output be correct. The copy constructor of a class is exactly the constructor that has the current class as a constant reference to the argument.

```
// https://godbolt.org/z/sGan39a8z
struct MyValue {
    static int counter;
    int number_;
    string name_;
    explicit MyValue(string name) // as before
        : number_{++counter} , name_{name}
    { cout << setw(number_) << '_' << "Constructor " << name_ << "\n"; }
    MyValue(const MyValue &orig) // new copy constructor
        : number_{++counter} , name_{orig.name_ + "-Copy"}
    { cout << setw(number_) << " " << "Copy Constructor " << name_ << "\n"; }
    ~MyValue() { // as before
        cout << setw(number_) << " " << "Destructor " << name_ << "\n";
    }
};
int MyValue::counter = 0;
```

Listing 16.14 The copy constructor takes a constant reference of the class as an argument.

Now if you run `main()` again, everything is back to normal. You see as many constructors as destructors:

```

    Constructor mvalue1
    Constructor temp
(function)
    Constructor local
    Destructor local
    Destructor temp
    Copy constructor mvalue1-copy
(function)
    Constructor local
    Destructor local
    Destructor mvalue1-copy
    Constructor mvalue2
    Destructor mvalue2
    Destructor mvalue1

```

Where you previously saw the first `Destructor mvalue1`, you see that in reality the previously invisible copy is being destroyed; this is revealed by the changed name in `Destructor mvalue1-copy`.

But even without the copy constructor, our program ran. If you do not define your own copy constructor, the compiler will once again take over for you. It generates a copy constructor that copies all data fields element-wise into the new instance. This works wonderfully with all data types you have encountered so far: the built-in types, standard containers, and most other types of the standard library.

However, it does *not* work if you use something that would require special handling in the destructor. Raw pointers and the `db_` resource from [Listing 16.7](#) would not be copied correctly (only the pointer, or the handle itself, would be transferred), and the resource would not be requested twice. And also our `cout ... <<"Destructor " ...` output falls into this category, as something special is done in the destructor. For a rule of thumb, see the following box.

When Do You Need a Copy Constructor?

If you write (or need to write) a destructor yourself to ensure your class functions correctly, then you very likely also need a self-written copy constructor—and vice versa. The resource you manage in the destructor very likely also requires special handling during copy and assignment.

If possible, you should try not to write a copy constructor yourself. The compiler not only takes care of the copy actions for you; it also does other important little things.

This includes automatic marking with `noexcept` if the data fields to be copied allow it. This can produce faster code in some cases.

16.7 Assignment Operator

You are now familiar with the `MyValue` class, and you also know that you can create new instances either with the `MyValue(string)` constructor or by copying with `MyValue(const MyValue&)`. You also know of the assignment with `=` and should therefore be able to say exactly what happens in the following examples.

```
void byVal(MyValue arg) { }
int main() {
    MyValue value1{"ABCD"}; // new instance, constructed via string
    MyValue value2{value1}; // new instance, constructed via copy
    MyValue value3 = value1; // new instance, also via copy, despite =
    byVal(value1);           // a new instance via copy
    value1 = value2;         // not a new instance, but an assignment
}
```

Listing 16.15 Copy and assignment.

While the first two lines contain no surprises, one might sometimes observe a raised eyebrow at `MyValue value3 = value1;`. Therefore, as a reminder: an equal sign `=` in a variable declaration is *never* an assignment, but just another way of writing initialization. To avoid confusion when reading, you should avoid this—at least when dealing with real classes. With simple types like `int`, `int val{7}` might look strange to some, so you can safely initialize with `=` here.

Now back to assignment. In `value1 = value2;`, you now have something new. Simplified once again, you have the following:

```
// https://godbolt.org/z/5M8xKabjK
int main() {
    MyValue value1{"ABCD"};
    MyValue value2{"WXYZ"};
    value1 = value2;        // assignment
}
```

You now get this:

```
Constructor ABCD
Constructor WXYZ
Destructor WXYZ
Destructor WXYZ
```

Oops, things have gotten mixed up again here. The Destructor WXYZ output appearing twice is puzzling. What happened here?

In `value1 = value2;`, the compiler calls the *assignment operator* of the `MyValue` class. You haven't defined an assignment operator yourself (as you are just learning it), so the compiler steps in for you once again and generates an assignment operator—and all data fields are element-wise assigned from `value2` to `value1`. What was previously contained in `value1` is then overwritten.

Thus, it is clear that one of the two Destructor WXYZ output lines would have been output without a Destructor ABCD assignment. And the indentation depth in the `number_` field was overwritten.

You probably want a different behavior; for example, you may want to keep the indentation depth of the original. To do this, implement a special method with the following signature:

```
MyValue& MyValue::operator=(const MyValue&);
```

Note that this is a *method* and not a *constructor*. The object to which something is assigned already exists; its content is to be overwritten by this assignment operator. Because it is not a constructor, you do not perform the transfer of the data fields in an initializer list, but like a method in the body.

```
// https://godbolt.org/z/s65GvY9xj
struct MyValue {
    // ... everything else as before
    MyValue& operator=(const MyValue& right) {
        if(this != &right) { // 1. check for self-assignment
            // 2. Release previous resources; none here
            // 3. element-wise transfer by assignment or similar
            name_ = right.name_ + "-Assignment (previously " + name_ + ")";
            /* number_ remains, and thus the original indentation */
        }
        return *this; // 4. return self
    }
};
```

Listing 16.16 The schema for implementing a custom assignment operator.

If you insert this into the definition of `struct MyValue`, you will get the following output:

```
Constructor ABCD
Constructor WXYZ
Destructor WXYZ
Destructor WXYZ-Assignment (previously ABCD)
```

And so the world is back in order: You can see from the text and indentation which original was overwritten and when it was cleared away.

If you define the `operator=` assignment operator for your class yourself, be sure to follow the implementation pattern shown previously:

1. Check with `if(this != &right)` whether this has the same address as `&right`. If you don't do this, problems can arise with self-assignments like `value1 = value1`.
2. Release all previous resources of the object to be overwritten. This is indicated in [Listing 16.16](#) at 2. because there is nothing to do for `MyValue` here.
3. Transfer all elements from `right` to left as under 3.. This is often done by assignment or some other form of copying. Because you intentionally omitted the transfer of `number_` in this case, add a comment about the unusual nature.
4. Then return a reference to the current object with `return *this`. This allows assignment chains like `value1 = value2 = value3 = value4;`.

The similarities of the assignment operator to the copy constructor are very extensive:

- Normally, you do not need a self-defined assignment operator; the compiler-generated one is at least as good for built-in data types, standard containers, and most types of the standard library.
- If you define (or need to define) a copy constructor or destructor yourself, you usually also need your own assignment operator—and vice versa.

Besides the different way you implement the assignment operator, there is another crucial difference: if you have `const` data elements in your class, then the assignment operator is removed from the list of available operations—or “deleted.” The consequence is that you can no longer overwrite an instance by assignment. To illustrate, I reduce the `MyValue` class to a constructor and a constant data field.

```
struct MyNumber {  
  
    const int number_;           // constant data field  
  
    explicit MyNumber(int v)  
        : number_{v}           // initialization of the constant data field  
        {}  
};  
  
int main() {  
    MyNumber c1{4};  
    MyNumber c2{7};  
    c1 = c2;                   // ✗ Error: Assignment removed by compiler  
}
```

Listing 16.17 The “const” data element is initially without an assignment operator.

The `c1 = c2` assignment is not possible here. The compiler tries to generate an assignment operator; it tries to assign `number_ = other.number_`. This is not possible because `number_` is declared with `const`. So instead of generating an assignment operator, the compiler *deletes* it. This results in the `=` operator not being available for this class.

16.8 Removing Methods

This has the same effect as if you had defined `MyNumber` as shown in the next listing. Note that I have again made `number_` non-`const`, and the compiler would have generated an assignment operator again.

```
// https://godbolt.org/z/398GhcbE
struct MyNumber {
    int number_;           // variable data field

    explicit MyNumber(int v)
        : number_{v} {}

    MyNumber& operator=(const MyNumber&) = delete; // remove assignment
    MyNumber(const MyNumber&) = delete;           // remove copy
};

int main() {
    MyNumber c1{4};
    MyNumber c2{7};
    c1 = c2; // ✖ Error – assignment removed by programmer
    MyNumber c3{c1}; // ✖ Error – copy removed by programmer
}
```

Listing 16.18 Use “`= delete`” to manually remove operations.

The `= delete` is a way for programmers to prevent the compiler from generating a method. And as you can see from the second `= delete` in `MyNumber(const MyNumber&) = delete;`, you can also remove a constructor with it. Thus, a copy of `MyNumber` in `MyNumber c3{c1};` is also no longer possible.

In our `MyValue` examples, it wasn't too problematic that I gradually addressed the necessary implementation of the copy constructor and assignment operator. The copy constructor or assignment operator generated by the compiler at worst produced incorrect output on the screen.

Listing 16.7 did not address copying and assignment, which resulted in an error-prone class. What happens if you copy or assign a `Database` instance? You end up duplicating a database handle or overwriting an existing one:

- If you *duplicate* it, then the destructor will call `db_close` on the same handle twice.
- If you *overwrite* it, then you miss a `db_close`, and a handle is not released.

Both are serious errors that occur if your listing does not adhere to the following rule: If you define a destructor yourself, you almost always need your own copy constructor and your own assignment operator as well.

With `= delete`, you can quickly get the problem under control for Database.

```
// https://godbolt.org/z/vb5a3EjcY
#include <iostream>           // cout
#include "database.hpp"       // include the foreign API

class Database {
    const db_handle_t db_;           // make constant
public:
    explicit Database(const char* filename);
    ~Database();
    int execute(const char* query);
    Database(const Database&) = delete;           // prohibit copying
    Database& operator=(const Database&) = delete; // prohibit assignment
};
// ... Implementations as before ...
int main() {
    Database db{ "customers.dat" };
    std::cout << "Count: " << db.execute("select * from cust") << "\n";
    Database db2{ db };           // ⚡ compiler prevents dangerous copy
    db = db2;                     // ⚡ compiler prevents dangerous assignment
}
```

Listing 16.19 With the deleted functions, the compiler prevents incorrect usage of the class.

I also have marked the `db_` data field with `const`. This alone would prevent the compiler from generating an `operator=`, making the program safe in this regard. Because I manually remove the copy constructor with `= delete` anyway, it doesn't hurt to play it safe and treat the assignment operator the same way.

16.9 Move Operations

You have learned two ways to copy a class instance. Let's assume your class is called `Image`:

- **`Image(const Image& other)`—the copy constructor**
This can only happen during the initialization of a new object.
- **`Image& operator=(const Image& other)`—the assignment operator**
This overwrites and only happens with an already existing object.

In most cases, all data fields from the source are copied into the target. Of course, this can be very costly.

```

// https://godbolt.org/z/75Kh5Ws45
#include <vector>
class Image {
    std::vector<std::byte> data_;
public:
    explicit Image(const char *fn) { /* ... */ }
    // Compiler generates (among others):
    // Copy constructor, assignment, but also moves
};
std::vector<Image> loadCollection(bool empty) {
    if(empty) return std::vector<Image>{};
    std::vector<Image> result {}; // for return; initially empty
    // three images in the collection ... copy?
    result.push_back( Image{"MonaLisa.png"} );
    result.push_back( Image{"TheScream.png"} );
    result.push_back( Image{"BoyWithPipe.png"} );
    return result; // return collection by value
}
int main() {
    // store return in variable
    std::vector<Image> collection = loadCollection(false);
}

```

Listing 16.20 The class probably contains large amounts of data that are expensive to copy. But what is being copied here?

I have written a sample data type `Image` that likely holds large amounts of data in `data_`. The compiler-generated operations for copy and assignment will completely copy this data field from the source to the target. So the data is duplicated after copying: I expect that, and thus it is fine.

Does it surprise you, then, when I tell you that at no point in [Listing 16.20](#) are image data duplicated in `data_`? If not, then you either have already read something about move operations or expect too much from C++. I would like to illustrate where duplicate data could exist due to copying, and then explain why this is not the case here:

- With `Image{"MonaLisa.png"}` , I create a new `Image` including its data. The `push_back` of `vector` takes this `Image` and places it where it stores its own data. This is a good candidate for a complete copy.
- With `return result;` , I return the vector by value. What was previously a local variable is to be passed outside. In this case, it should be stored in another variable with `collection = ....` Such a scenario is another very good candidate for a copy—which here would be the copy of all `vector-Image` elements.

A vector has its own area where it stores its data. The new `Image` instances within `loadCollection()` are, of course, unaware of this and certainly store their data elsewhere. The `push_back` method of `vector` copies this data to where the vector wants it. The good news is that this copy can be omitted under certain circumstances—as in this example, `vector` can save the expensive copying of data for objects designed with `Image`. To achieve this, it is—as always—sufficient to use only built-in types, standard containers, or most types from the standard library, and then to not define a copy constructor, destructor, or assignment operator.²

After `return result;`, it no longer makes sense to keep `result` as a local variable. When leaving the function, `result` is discarded; the destructor is called, and all contents are destroyed. It is good that the content was copied to `collection` beforehand for the return. Should you first make a complete copy and then destroy the original? This is not necessary in C++ either. In the case shown here, the content of `result` can be directly transferred—moved—to `collection`. This effect during return is called *copy elision*.

Neither is self-evident and both are subject to certain conditions. C++ can in no way forgo calling the destructor of `result` as that would contradict the RAII principles. And when the destructor is called, you can expect the data fields to be removed as well. The solution to the puzzle is that a *special form of copying* is applied—not the one you already know, but one that only works with temporary values.

The `&&` symbol here denotes a special form of reference, the *temp-value reference* (technically, the *rvalue reference*). For the moment, please consider this like the normal reference `&`, but remember that when dealing with moving you need `&&`:

■ **`Image(Image&& other)`—the move constructor**

Like the copy constructor, except that `other` is not `const`. Therefore, its data can be taken away from it. Used by the compiler only when `other` will not exist for much longer, which is why taking away the data is acceptable.

■ **`Image& operator=(Image&& other)`—the move assignment operator**

Like the assignment operator, except that `other` is not `const`. Data is taken away from it. The compiler automatically uses this operation when it is sure that `other` will be destroyed soon.

In [Listing 16.20](#), the compiler uses both to avoid copies:

- `vector` must first create an empty object with `Image{}` in `push_back`. Containers use this default creation without arguments when they need it. The newly created `Image` is then assigned using the move assignment operator.
- `vector<Image> collection = ...` is initialized with the move constructor. That means the return value, and thus `result`, takes the data instead of copying the entire object.

² At least for types that can be moved, e.g., that don't have reference or pointer members. Also, some types of the standard library have special copy or move semantics, like `unique_ptr` or `mutex`.

How do the two operations you just learned about do this? The matter becomes clear when you consider the following facts:

- The compiler ensures that `other` is short-lived and will be destroyed soon anyway.
- Therefore, `other` no longer needs its data, and they can serve a better purpose—like avoiding a copy of the data.
- The object that is supposed to receive the data is usually freshly created—almost empty.
- The receiving object is supposed to have its data overwritten; it no longer needs its old—mostly empty—data.
- Both involved objects must be valid before the operation. `this` is at least initialized empty; `other` contains the valuable data.
- Both objects must be valid after the operation. `this` contains the data; `other` can be empty, ready for the destructor call.

If you had to implement the operations yourself with these requirements—which you don't have to, as the compiler does it—you could do it as in the following listing.

```
// https://godbolt.org/z/aoqW86vs8
#include <vector>
class Image {
    std::vector<std::byte> data_;           // byte has been available since C++17
public:
    explicit Image(const char *fn) { /*...*/ }
    Image(Image&& other) noexcept          // move constructor
        : data_{} // create empty
    {
        using std::swap;
        swap(data_, other.data_);
    }
    Image& operator=(Image&& other) noexcept { // move operator
        using std::swap;
        swap(data_, other.data_);
        return *this;
    }
};
```

Listing 16.21 Implementation of the two move operations.

With `swap`, you exchange almost any element in the most efficient way imaginable. Normally, you can also directly use `std::swap`, but it has become an idiom that a prior using `std::swap` leaves the possibility open to use an even better `swap` variant. More important, however, is the question of what is being swapped here. Actually, everything has already been said with the preceding bulleted list: before swapping, `other`

contains the data, afterward this does; before, this is (mostly) empty, afterward it is other. Assuming that `swap` does not copy any data, you have very effectively transferred the data from other to this and left both objects in a correct state.

The `swap` function is useful, but in some cases, the similar `exchange` function is more appropriate:

```
Image(Image&& other) noexcept           // move constructor
: data_{ std::exchange(other.data_, {})}
{ }
```

Effectively, `r = exchange(old, new)` performs `r=old` and `old=new` sequentially where possible, using `move`. Thus, you can usually initialize all members in the move constructor.

If you do not write a copy constructor, assignment operator, or destructor because you do not need them, then you usually do not need either of the move operations. In that case, the compiler generates the move operations for you.

Or the compiler prohibits moving, similar to how it would prohibit copying. For example, if you have a `const` data field, it cannot be moved from.

If you manage to create a movable data type, it is ideal for return values and for being put into standard containers. The compiler ensures that sorting, swapping, pushing, and resizing use the move operations. The noble goal of a data type that is to be put into a container is to be movable. You can achieve this in two ways:

- Give your class only data fields that are themselves movable—for example, built-in types, standard containers, or types from the standard library.
- You take on the thankless task of implementing the move operations yourself.

You should not do the latter, really. And if you do, then let me tell you that you must also write `noexcept` after those operations—such a profound topic that I have only touched on it in this book (see [Chapter 7](#)).

16.9.1 What the Compiler Generates

It is important to understand when the compiler takes over the work of generating a constructor or `operator=` for programmers and when it does not. The rules are not trivial, but they are written in such a way that your program becomes as safe and fast as possible automatically.

Here, *automatically generated* means that the compiler implicitly declares the special function in question as if you had marked it with `= default`:

- **The default constructor**
`C()` is automatically generated if you do *not declare any* constructor yourself.
- **The destructor**
`~C()` is automatically generated if you do not declare a destructor.

- **The copy constructor**

`C(const C&)` is automatically generated if you do not declare either of the move operations.

- **The copy assignment**

`C& operator=(const C&)` is automatically generated if you do not declare either of the move operations.

- **The move constructor**

`C(C&&)` is automatically generated if you do not declare either of the copy operations and all elements can be moved.

- **The move assignment operator**

`C& operator=(C&&)` is automatically generated if you do not declare either of the copy operations and all elements can be moved.

From this, you can conclude that the copy operations are *independent* of each other: if you declare one of them, the compiler could still generate the other. So, for example, if you declare the copy constructor but not the copy assignment, and then write code that requires `C& operator=(const C&)`, the compiler will generate it.

This does not apply to move operations; these are *not independent* of each other. If you declare one of the two, the compiler will not generate the other.

16.10 Operators

You have now learned about the special `operator=` method, which is called by the compiler when you write the following:

```
Data data{};
Data newData{};
data = newData;
```

The equal sign `=` of the assignment is also just a *binary operator* with a right operand and a left operand. It is called *infix notation* when the operator symbol is between the two operands. The compiler translates this into the following method call:

```
data.operator=(newData); // method notation
```

Although `operator=` is not a normal identifier, it is allowed in this usage form—but only if you have actually defined such a method.

However, you have also redefined another operator: the `operator<<` output operator. To output `Year` to an ostream, you have overloaded a free function:

```
std::ostream& operator<<(std::ostream& os, const Year&);
```

If you do not want to use the infix notation, you can use the function notation analogously to the method notation:

```
cout << year;           // infix notation
operator<<(cout, year); // function notation
```

If you use the infix notation, the compiler automatically selects the method or free function, depending on what you have defined for your class. If you use the function or method notation, you choose it yourself. This can be useful in cases of doubt, for example, if you have both.

```
// https://godbolt.org/z/P99563KoP
#include <iostream>
using std::cout; using std::ostream;
struct Widget {
    bool operator<(const Widget&) const {    // method notation
        return true;                       // always true
    }
};
bool operator<(const Widget&, const Widget&) { // function notation
    return false;                           // always false
}
int main() {
    Widget x{};
    Widget y{};
    cout << (operator<(x, y)           // calls function notation
        ? "Method1\n" : "Function1\n");
    cout << (y.operator<(x)           // calls method notation
        ? "Method2\n" : "Function2\n");
    cout << (x < y                     // infix notation, allows choice, here function
        ? "Method3\n" : "Function3\n");
}
```

Listing 16.22 Infix, function, and method notation for operators.

For the `operator<` of `Widget`, two variants are implemented: the free function and the method. To observe the difference in invocation, the two implementations return different values. The first two callers in `main` explicitly choose the respective implementation: `operator<(x, y)` calls the free function, and `Function1` is printed. `y.operator<(x)` is the method call and results in `Method2`. With the infix notation `x < y`, the compiler can choose. Here, GCC prefers the function, and `Function3` is printed, while Clang++ produces an error.

As you saw in [Chapter 4, Section 4.3](#), there are more operators. And just like `operator=`, you can define most of them for your class yourself. But how? As a free function or as a method? Most of the time, you can choose whether to define an operator for your own

classes as a method of the class or as a free function. In practice, however, one or the other has become established for each operator. The following rules of thumb will help you choose:

- Unary operators should be implemented as methods. This includes `operator++` and `operator--`, but also `!`, `~`, and the unary forms of `+`, `-`, and `*`.
- Binary operators that do not modify any of the arguments should be defined as free functions. These are primarily the bitwise and arithmetic operators, including `<<` and `>>`, which are used for input and output. Comparison operators are implemented via `<=>` and optionally also `==`.
- Binary operators that modify the left argument are best suited as methods—for example, `operator+=`.
- Assignment, index operator, and call operator must be methods—that is, `operator=`, `operator[]`, and `operator()`.

There are a few more operators that I won't mention here because they are only used in special cases. These are just general guidelines.

A disadvantage of implementing as a free function is that it does not give access to the private data fields and methods of the class the function is supposed to work on. However, often with these operators, delving into the internals is helpful or necessary. Then you have three options:

- You can take a detour via a public method that offers a similar function—as seen in [Chapter 12, Listing 12.27](#) with `operator<<` and `Year::print`.
- You can write the operator as a friend within the class.
- The worst way would be to make the private internals public, as this contradicts encapsulation.

Whether you choose the first or second approach depends on the circumstances.

A friend function is a special notation of a free function that you write *inside* a class.

```
// https://godbolt.org/z/Wvr3aK75G
#include <compare>           // for <=>
class Value {
    int value_;
public:
    explicit Value(int value) : value_{value} {}

    // free functions, but declared as friend
    friend Value operator+(const Value& li, const Value& re);
    friend Value operator-(const Value& li, const Value& re) {
        return Value{li.value_ - re.value_}; // also defined inside
    }
}
```

```
// operator<=> does not need to be friend.
// for all comparisons: <, >, <=, >= as well as == and !=
auto operator<=>(const Value& re) const = default;
};
// Definition of the previously declared friend function:
Value operator+(const Value& li, const Value& re) {
    return Value{li.value_ + re.value_}; // access to private members allowed
}
int main() {
    Value seven{7}; Value three{3}; Value two{2};
    if((three+two) < seven) { // operator+, then operator< via <=>
        return 0;           // okay
    } else {
        return 1;           // something went wrong
    }
}
```

Listing 16.23 A “friend” function is not a method, even if it is inside the class or defined there.

Even if the functions, here `operator+` and `operator-`, are now inside the `Value` class, they are free functions—as if you had written them outside. The differences are as follows:

- In the function body, you can access everything you could access from a method.
- Other free functions can be added to your data type by anyone as they wish. For friend functions, however, the following applies: because this special form is written within the class definition, you as the class designer determine which free friend functions exist.

For `operator+`, I have separated the function declaration within the class from the function definition outside—as is also common with a normal method. When defining, you do not repeat the `friend` keyword. At that point, you do not see that it is a friend and why it suddenly has private access rights. Perhaps that is why it is quite common for friend functions to also perform the definition within the class, as was done with `operator-`. Of course, you should only consider this for short functions.

The `operator<=>` is a special case because it does not need to be declared as a friend function. More on this is to come in the next section.

16.11 Custom Operators in a Data Type

Many operators can theoretically be implemented in a variety of ways. And you are almost completely free to choose the argument and return types. For a `Value` class, you can certainly have `operator++` return `void`, as it is practical to be able to use the incremented value immediately.

Meaning over Convenience

When adding operators to your class, you should avoid creating surprises. Maintain the approximate meaning of the operator symbol. Let's take this code as an example:

```
Image img1{}; Image img2{};
auto x = img1 + img2;
```

Nothing would be more confusing than if the expression `img1 + img2` did not merge the two images but, for example, displayed them sequentially on the screen. It is better to use descriptive function or method names.

Personally, I follow the rule that I only define operators if I use the operation in expressions and the readability of the overall expression increases.

Here, I will show you the most common argument and return types of most operators and indicate whether it is common to implement them as member functions or free functions. Because there are quite a few operators, [Listing 16.24](#) provides a longer example.

I have defined most operators directly in the class, but some, mainly for space reasons, outside the class. By the way: you should also mark such short implementations of operators outside the class with the `inline` keyword. The compiler then strives more for speed. Users often expect performance from operators. When implemented within the class, the method is implicitly `inline` for the compiler. In both cases, you can also consider `constexpr`, which allows computation at compile-time and also implies `inline`.

There are some peculiarities to consider with operators:

- **Binary arithmetic operators return new objects**

All binary arithmetic and bitwise operators must always return a new object. Do not be tempted to trick with references or even pointers. Avoid hard-to-find errors by adhering to this rule.

- **Compound assignments as methods**

Because `+=` and its relatives modify the left operand, they are best suited as methods.

- **Implementing binary operators using compound assignments**

When you implement `a+b`, make it easy on yourself by also implementing `+=`. Then pass the left parameter by value—that is, `Num a` instead of `const Num& a`—and let the compiler create a copy. Apply `a += b` to this copy—and you can return its `Num` directly with `return`.

- **Overloading operators for argument types**

For the operator `+=`, I have defined two overloads. One takes a `const Num&` as a parameter, the other an `int`. Normally, you should prefer the current class as the parameter type, as it is the most natural. However, you would have to write `a += Num{7}`. Where it makes sense, you can add overloads and allow `a += 7`;—that is, with an `int`.

- **Comparisons via `<=>` or individually as free functions**

Since C++20, there is the *spaceship operator* `<=>`, which implements all comparisons at once. If you define it, the compiler derives all comparisons `<`, `<=`, `>`, and `>=` from it. If you cannot or do not want to use it, you should implement the operators as free functions with two `const` references.

- **If you define `<=>` yourself, then also define `==`**

If you define `operator<=>(...) = default`, the compiler will also derive `==` and `!=` from it. If you implement it yourself, you should also define `operator==` yourself.

- **Ordered types implement the `<` and `==` operators**

Your class is equipped to handle sorting standard containers like `map` and `set` if you only implement `<` and `==`. But you can also go so far as to offer all comparison operators. And then it is easiest if you define `<=>`.

- **`==` implies `!=`**

Since C++20, it is sufficient if you implement `a == b`. If the compiler needs `a != b`, it will try `!(a == b)`. If `a` and `b` are of different types, it will also try `b != a` and `!(b == a)`. If you do not want this, then define the corresponding overloads yourself.

- **Use `constexpr`, `noexcept` or `const`**

If you can, mark your operators with `constexpr` and `noexcept`. Member functions should at least be marked with `const`.

- **Preincrement and predecrement**

`operator++()` and `operator--()` modify the object immediately and then return it as a reference. This allows, for example, `++a`. If you use `a++` instead (postincrement), then a copy of the old value needs to be created and returned; you should avoid this and prefer the pre variants. If you still want to provide the post variants, you need to define `operator++(int)` and `operator--(int)`. The `int` parameter is just a dummy and is not used; it only serves to distinguish the overload. The implementation would look like this:

```
//The less efficient a++ and a--
Num Num::operator++(int) { Num res{*this}; val_++; return res; }
Num Num::operator--(int) { Num res{*this}; val_--; return res; }
```

- **Dereference operator**

The unary `operator*` is typically used to access the “real value” of a class. Good examples of this are `unique_ptr`, all iterators, and raw pointers. So if you write `Num a{7}`;, you can now write `*a` to access `val_`. Because I also return a reference with `int&`, you can even do `*a = 99`; to change the value.

```
// https://godbolt.org/z/YonbGKj8M
#include <iostream> // istream, ostream, cout
class Num {
    int val_ = 0;
public:
```

```

int& operator*(); // Dereference: Direct access to the value
const int& operator*() const; // Dereference: Read access to the value
Num() {}
explicit Num(int value) : val_{value} {}

// unary operators
Num& operator++(); // Preincrement
Num& operator--(); // Predecrement
Num operator+(); // Positive
Num operator-(); // Negate
Num operator~(); // Bitwise invert

// binary operators
// - compound assignments, arithmetic
Num& operator+=(const Num& re) { val_ += *re; return *this; }
Num& operator-=(const Num& re) { val_ -= *re; return *this; }
Num& operator*=(const Num& re) { val_ *= *re; return *this; }
Num& operator/=(const Num& re) { val_ /= *re; return *this; }
Num& operator%=(const Num& re) { val_ %= *re; return *this; }
// - compound assignments, bitwise
Num& operator|=(const Num& re) { val_ |= *re; return *this; }
Num& operator&=(const Num& re) { val_ &= *re; return *this; }
Num& operator^=(const Num& re) { val_ ^= *re; return *this; }
Num& operator<<=(int n) { val_ <<= n; return *this; }
Num& operator>>=(int n) { val_ >>= n; return *this; }
// - Variation of compound assignments for easier handling
Num& operator+=(int re) { val_ += re; return *this; }
Num& operator-=(int re) { val_ -= re; return *this; }
// binary operators, with call-by-value for the first parameter
// and using compound assignment for assistance
// - Arithmetic
friend Num operator+(Num li, const Num& re) { return li += re; }
friend Num operator-(Num li, const Num& re) { return li -= re; }
friend Num operator*(Num li, const Num& re) { return li *= re; }
friend Num operator/(Num li, const Num& re) { return li /= re; }
friend Num operator%(Num li, const Num& re) { return li %= re; }
// - bitwise
friend Num operator|(Num li, const Num& re) { return li |= re; }
friend Num operator&(Num li, const Num& re) { return li &= re; }
friend Num operator^(Num li, const Num& re) { return li ^= re; }
// - comparisons
// - ... fundamental for standard containers and algorithms
friend bool operator==(const Num& li, const Num& re) { return *li == *re; }
auto operator<=>(const Num& re) const {return val_ <=> *re; } // C++20
// - input and output
friend std::ostream& operator<<(std::ostream& os, const Num& arg);

```



```

    friend std::istream& operator>>(std::istream& is, Num& arg);
};

// unary operators
Num& Num::operator++() { ++val_; return *this; }
Num& Num::operator--() { --val_; return *this; }
Num Num::operator+() { return Num{val_}; }
Num Num::operator-() { return Num{-val_}; }
Num Num::operator~() { return Num{~val_}; }
int& Num::operator*() { return val_; }
const int& Num::operator*() const { return val_; }

// Input and Output
std::ostream& operator<<(std::ostream&os, const Num& arg) { return os<<*arg; }
std::istream& operator>>(std::istream&is, Num& arg) { return is>>*arg; }

```

Listing 16.24 A data type almost fully equipped with all operators.

Note that it is not necessary to declare `operator<=>` as a friend. You could also implement it as a free function with two parameters. This will automatically implement `<`, `<=`, `>`, and `>=`. My implementation is identical to `=` default—which would have also saved me from implementing `operator==`. `!=` is derived from `==`.

I have omitted `constexpr` and `noexcept` everywhere to avoid impairing readability. Now you can use `Num` as you are accustomed to.

```

// https://godbolt.org/z/YonbGKj8M
#include <iostream> // cout
int main() {
    using std::cout;
    Num a{1};
    *a = 7; // operator* also returns int&
    a += Num{3}; // Increment with Num
    cout << ( ++( ++a ) ) << "\n"; // Output: 12
    a -= 2; // Variation with int
    cout << --(--a) << "\n"; // Output: 8
    Num b{99};
    cout << (a<b ? "yes\n" : "xxx\n"); // Output: yes
    cout << (a>b ? "xxx\n" : "no\n"); // Output: no
    b /= Num{3}; // b: 33
    b %= Num{10}; // b: 3
    b <<= 4; // b: 48
    b >>= 2; // b: 12
    Num c = b / Num{3} + a * Num{2}; // c: 20
}

```

Listing 16.25 You use the data type equipped with operators as usual.

According to these principles, you can also implement the operators for Boolean logic in a more suitable data type.

```
// https://godbolt.org/z/vjrr6d5TW
#include <iostream> // istream, ostream, cout
class Bool {
    bool val_ = false;
    bool& operator*() // dereference; mutable
    { return val_; };
    const bool& operator*() const // dereference; read-only
    { return val_; }
public:
    constexpr Bool() {}
    explicit constexpr Bool(bool value)
        : val_{value} {}
    // unary operators
    Bool operator!() const // not operator
    { return Bool{!val_}; };
    // binary operators
    friend Bool operator&&(const Bool &re, const Bool &li)
    { return Bool{*re && *li}; }
    friend Bool operator||(const Bool &re, const Bool &li)
    { return Bool{*re || *li}; }
    // input and output
    friend std::ostream& operator<<(std::ostream& os, const Bool& arg);
    friend std::istream& operator>>(std::istream& is, Bool& arg);
};
std::ostream& operator<<(std::ostream& os, const Bool& arg)
{ return os << *arg; }
std::istream& operator>>(std::istream& is, Bool& arg)
{ return is >> *arg; }
// Constants
static constexpr Bool False{false};
static constexpr Bool True{true};
int main() {
    Bool yesno = True && ( Bool{false} || !Bool{} ); // uses &&, ||, and !
    std::cout << yesno << "\n"; // Output: 1
}
```

Listing 16.26 This data type demonstrates the Boolean operators.

The unary not operator `!` is also implemented as a method, and the binary logical operators `&&` and `||` as free functions.

I also define the `True` and `False` constants so that I can use them like literals in expressions more easily. To be able to do this everywhere, it is advisable to use the consistent declaration with `constexpr` and do the same for the constructors of `Bool`.

The `constexpr` on the constructor makes `Bool` a *literal data type*. This allows you to use `Bool` in other `constexpr` expressions as well. You can learn more about this in [Chapter 18, Section 18.6](#).

I should go into more detail about `operator<=>`. Here is the definition again:

```
constexpr auto operator<=>(const Num& re, const Num& li) {  
    return *re <=> *li;  
}
```

Instead of letting the compiler deduce the return type with `auto`, you can also specify an explicit return type. This must be one of the *comparison category types* from the `<compare>` header. If you now write `operator<=>` with an explicit return type, the predefined constants in the corresponding category are suitable:

- **strong_ordering with less, equal, greater**
You use this when every value of your type can be compared with every other value of your type—including itself—as less, equal, or greater. A typical example of this is `int`; any `int` values can always be compared.
- **weak_ordering with less, equivalent, greater**
Like `strong_ordering`, but some values are *equivalent* instead of *equal*. Strings, when case is ignored, are an example.
- **partial_ordering with less, equivalent, greater, unordered**
Like `weak_ordering`, but there can be values of the type that cannot be compared at all. Floating-point numbers with NaN are an example.

This might look like the following listing for your data type.

```
// https://godbolt.org/z/KxWb4q8q9  
#include <compare> // partial_ordering etc  
#include <set>  
#include <iostream>  
using namespace std;  
struct Fraction {  
    int z, n; // numerator / denominator  
    partial_ordering operator<=>(const Fraction& rhs) const {  
        if(n==0 || rhs.n==0) return partial_ordering::unordered;  
        return (double)z / n <=> (double)rhs.z / rhs.n;  
    }  
};
```

```
int main() {
    set<Fraction> fractions{ {1,2},{2,4},{1,3},{2,3},{1,4},{2,5},{3,8},{99,0} };
    for(auto f : fractions)
        cout << f.z << "/" << f.n << " ";
    cout << "\n"; // Output: 1/4 1/3 3/8 2/5 1/2 2/3
}
```

Listing 16.27 The comparison categories contain constants for the return.

I know, my fraction is not really competitive compared to `std::ratio`.

As you can see, the output is sorted by size. The incomparable element ends up at the end. You also see that 2/4 is missing in the output because it corresponds to 1/2. The fraction 99/0 does not appear at all due to incomparability.

It is best to rely on the built-in overloads of `<=>`, as shown here with `return`.

You can compare the return of `operator<=>` with zero to see if the comparison resulted in equal, less, or greater. For example, you can implement comparison chains with multiple criteria:

```
auto operator<=>(const Address& re) const {
    auto cmp = street <=> re.street;
    if(cmp != 0) return cmp; // not equal, result is clear

    cmp = zip <=> re.zip;
    if(cmp != 0) return cmp; // not equal, result is now clear

    return city <=> re.city;
}
```

For this case, there is also the practical `lexicographical_compare_three_way(a,b)` algorithm.

```
auto operator<=>(const Address& re) const {
    array a{street, zip, city};
    array b{re.street, re.zip, re.city};
    return lexicographical_compare_three_way(begin(a), end(a), begin(b), end(b));
}
```

Listing 16.28 You can use this helper function to implement a simple ordered comparison.

The `lexicographical_compare_three_way` algorithm takes two pairs of iterators, which is why I create two arrays here that contain elements to compare.

Instead of `a <=> b`, there is also the `compare_three_way(a,b)` functor, which in most cases does the same thing but may handle pointers better.

16.12 Special Class Forms

In C++, there are special class forms that help you structure your code. Abstract classes and methods allow you to define an interface first without implementing it. Enumeration classes, in turn, bundle possible values of a class.

16.12.1 Abstract Classes and Methods

You learned about virtual methods in [Chapter 15](#). With these, you can do something very special—namely, methods *without* implementation.

```
// https://godbolt.org/z/1zq4jfdYb
#include <string>
#include <iostream>
using std::string; using std::ostream;

class Shape {
    string color_;
public:
    virtual double calcArea() const = 0; // pure virtual method
    string getColor() const { return color_; }
    void setColor(const string& color) { color_ = color; }
    virtual ~Shape() {}
};

class Square : public Shape {
    double len_;
public:
    explicit Square(double len) : len_{len} {}
    double calcArea() const override { return len_*len_; }
};

class Circle : public Shape {
    double rad_;
public:
    explicit Circle(double rad) : rad_{rad} {}
    double calcArea() const override { return 3.1415*rad_*rad_; }
};

struct Calculator {
    Shape& shape_;
    Calculator(Shape& shape) : shape_{shape} { }
```

```

    void run(ostream& os) const {
        os << "The area of the shape is " << shape_.calcArea() << "\n";
    }
};

int main() {
    Square square {5.0};
    Calculator ti { square };
    ti.run(std::cout); // Output: The area of the shape is 25
}

```

Listing 16.29 A virtual method that is = 0 is called pure virtual or abstract.

The base `Shape` class knows that `calcArea` is supposed to calculate the areas of shapes. However, it does not know how. Therefore, the method is pure virtual, marked with = 0. Note that you should define a virtual destructor if you mark at least one method as virtual, even if you do not write any source code in the destructor. The compiler will insert cleanup code there. If you do not write `virtual` yourself, the destructor generated by the compiler will not be virtual.

The derived `Square` and `Circle` classes know how to calculate `calcArea` and therefore implement this method.

A class with at least one pure virtual method is called an *abstract class*. It is not possible to instantiate an abstract class directly. The compiler will respond to the attempt with an error:

```
Shape form {}; // ✗ abstract class
```

Classes that inherit from an abstract class must override all pure virtual methods if they do not want to be abstract themselves. Only then can you instantiate them.

In practice, users of the class then store a pointer or a reference to the abstract base class, just as `Calculator` does with `Shape& shape_`. A reference or pointer is necessary here to enable polymorphism. Without it, the value would be converted, and parts of it would be copied—which wouldn't even compile with abstract base classes.

As with polymorphic instances in general, you will often see that a base pointer is initialized with a concrete pointer:

```

Shape* shape1 = new Circle{5.0};
delete shape1; // don't forget, please
unique_ptr<Shape> shape2 = make_unique<Square>( 7.0 ); // safer

```

In [Chapter 20](#), you will also see that for a new for a raw pointer, the corresponding `delete` is of immense importance.

16.12.2 Enumeration Classes

You have learned that integer types like `int` and the string type `string` are very useful data types. For example, if you wanted to write a `Stoplight` class, you could use it to represent the current light. With an `int`, you would “remember” that 0 stands for red, 1 for yellow, and 2 for green. Or you could use a `string` and consistently use the literals “red”, “yellow”, and “green”. You can avoid typos by defining constants like `const string RED = “red”;`.

However, the ubiquitous `int` and `string` types are often referred to as *type sinks*—which roughly means “black holes for types.” Let’s assume you wrote a method that takes a traffic light color:

```
Stoplight createStoplight(string lightColor, string lightName);
```

Many things can go wrong here:

```
createStoplight("AX-001", YELLOW);           // ✖ Arguments swapped
createStoplight("yellowgreen", "AX-002");     // ✖ Undefined color used
createStoplight("grren", "AX-003");           // ✖ Typo in color
```

These are all errors that the compiler does not notice—and perhaps you don’t either, but only the customer does.

The solution is to write a `StoplightColor` *enumeration class* that defines the exact valid range—no more and no less.

```
// https://godbolt.org/z/xhd7s1YhE
#include <string>
using std::string;
enum class StoplightColor {
    RED, YELLOW, REDYELLOW, GREEN
};
struct Stoplight {
    StoplightColor color_;
    Stoplight(StoplightColor color, string name) : color_{color} {}
};

Stoplight createStoplight(StoplightColor color, string name) {
    return Stoplight{color, name};
}

int main() {
    Stoplight stoplight = createStoplight(StoplightColor::RED, "AX-001");
}
```

Listing 16.30 With an “enum”, you define a type with its own value range.

Now you can implement different behaviors depending on the traffic light color. This is often used in switch statements or if-else chains:

```
string drivingSchool(Stoplight light) {
    switch(light.color_) {
        case StoplightColor::RED:
            return "stop";
        case StoplightColor::REDYELLOW:
            return "get ready";
        case StoplightColor::YELLOW:
            return "brake";
        case StoplightColor::GREEN:
            return "go";
    }
    abort(); // exhaustive switch, should not happen
}
```

We've handled all possible cases in the switch, so the `abort()` should not occur. Alternatively, we could have defined a default: case.

It's good that with enum class, RED and YELLOW no longer enter the surrounding namespace, but having to always write the `TrafficLightColor::` prefix is cumbersome, especially in such a switch. Therefore, since C++20, there is the possibility to bring the enumeration class into the current namespace with `using enum TrafficLightColor;`

```
// https://godbolt.org/z/f8on7bs1Y
string drivingSchool(Stoplight light) {
    switch(light.color) {
        using enum StoplightColor; // "StoplightColor::" can now be omitted
        case RED:
            return "stop";
        case REDYELLOW:
            return "get ready";
        case YELLOW:
            return "slow down";
        case GREEN:
            return "go";
    }
    abort(); // exhaustive switch, should not happen
}
```

Individual values like using `StoplightColor::RED` can also be imported. This can be important to avoid name conflicts.

In other applications, the possible `enum` values are backed by specific numerical values. You can access these with a type conversion. You can specify the numerical values in the enumeration class.

```
// https://godbolt.org/z/Yjbzrrnob
enum class Weekday {
    MO=1, TU, WE, TH, FR, SA, SU           // TU becomes 2, WE becomes 3, etc.
};
enum class Level {
    TRACE=1, DEBUG, INFO=10, ERROR, FATAL // gaps are also possible
};
void log(Level level) {
    int intLevel = (int)level;             // explicitly cast to an int
    if(intLevel > 10) { /* ... */ }
}
```

Listing 16.31 In an “enum”, you can also specify the desired numerical values.

The C++ compiler assigns values incrementally after each explicitly specified value. Thus, in `Level`, `DEBUG` becomes 2, `ERROR` becomes 11, and `FATAL` becomes 12. If, as a user, you need the `int` value of an element when using it, you can obtain the value through explicit type casting like `(int)level`.

Chapter 17

Good Code, 4th Dan: Security, Quality, and Sustainability

In this chapter, I want to summarize some things that can particularly help you in the development of *good* software. Throughout the book, you will find many more tips, and, especially in [Chapter 29](#), a long collection, but each is only briefly touched upon.

The special thing about the tips in this chapter is that they are best applied *from the very beginning*. If you incorporate the ideas *behind* the rules into your daily programming routine, your perspective on C++ and your thinking with C++ will change in such a way that better code will automatically emerge.

17.1 The Rule of Zero

A class can be equipped with many functions for special purposes. You can initiate an automatic conversion, you can specify how a class should be copied, and so on.

Read the following paragraphs at ease—because at the end I will tell you that you should not implement any of these functions.

17.1.1 The Big Five

Pay special attention to the following special functions:

- **Destructor** `~Type()`

The compiler generates a destructor for you that cleans up member variables but does not call `delete` for raw pointers or `delete[]` for C-arrays (see [Chapter 20](#)). If your class has such fields, you need to write a destructor.

- **Copy constructor** `Type(const Type&)`

If you do not write a copy constructor yourself, the compiler generates one. However, this only copies all fields. This is bad for raw pointers, for example, because then two objects point to the same memory object; it looks like both “own” the object. And if you (rightly) wrote your own destructor, then both will call `delete` on the same memory object. You need to write a copy constructor yourself that copies the *content* of pointers.

- **Assignment operator** `Type& operator=(const Type&)`

If you do not write the assignment operator yourself, but raw pointers belong to the class, then the compiler generates an assignment that simply overwrites the current pointers without deleting them first. In addition, after the assignment, both objects point to the same memory area: doubly bad. Therefore, with raw pointers, you need to intervene yourself and copy the content again.

- **Move constructor** `Type(Type&&)`

Again, raw pointers are a problem here. The code the compiler generates for raw pointers in a move constructor just copies the pointer itself. This would mean two pointers to the same data. If you really want to move, you have to implement it yourself. If the compiler doesn't generate the copy constructor, it also doesn't generate the move constructor.

- **Move operator** `Type& operator=(Type&&)`

Just like with the move constructor, a move operator generated by the compiler does not handle raw pointers satisfactorily. You have to take matters into your own hands. Also, if the compiler doesn't generate the assignment operator, it also doesn't generate the move operator.

It is said that if you need to implement even one of these five functions, then you must implement the others as well. This is the *rule of five*. This is because all these functions need to handle unclear ownership scenarios. And if you need to manage such ownership in one of the functions, then you must do so in the others as well.

You may have noticed that *what the compiler can generate* was mentioned each time. And each time, raw pointers and dynamically allocated C-arrays are the member variables that need special attention.¹

17.1.2 Helper Construct by Prohibition

Instead of tackling the not-so-simple task of implementing all these functions, you can at least *prohibit* them. If you disable all copy and move operations, then ownership cannot get mixed up.

```
// https://godbolt.org/z/K1714eWr9
struct Type {
    char* data_;           // raw pointer can cause unclear ownership
    explicit Type(int n) : data_(new char[n]) {}
    ~Type() { delete[] data_; } // you need the destructor

    Type(const Type&) = delete;           // do not allow copying
    Type& operator=(const Type&) = delete; // no assignment please
```

¹ These are not the only ones, but the most common and important.

```

    Type(Type&&) = delete;           // no moving
    Type& operator=(Type&&) = delete; // no move assignment operator
};

```

Listing 17.1 Prohibit four of the big five with “= delete”.

So, if you unfortunately have a type that has a problematic field (which you cannot or do not want to get rid of), then for your safety, your first step should be to prohibit accidental copying and moving, as shown in [Listing 17.1](#).

By using = delete after critical operations, you prevent the compiler from generating inappropriate functions.

17.1.3 The Rule of Zero and Its Application

Am I scaring you away from using raw pointers? Good! Because the solution to the puzzle, the philosopher's stone, the holy grail is this: take advantage of the compiler's ability to generate functions for you. You just need to allow it to do so correctly.

This is a dan chapter, and you won't learn how to handle raw pointers until [Chapter 20](#). But as I want to discourage you from using raw pointers anyway, this preview has its advantages. First, memorize the alternatives. However, you will encounter one or another raw pointer in the following paragraphs.

The most important thing is that you do not use raw pointers that *own* data—that is, having requested them with `new` and thus being implicitly responsible for calling `delete`. Instead, use what the standard library offers you:

- For large amounts of data, there are containers. You no longer need to build linked structures yourself; you have a wide selection.
- Every `new` should only go directly into a smart pointer. Better yet, use `make_shared` and `make_unique`.
- With smart pointers, you no longer need `delete` or `delete[]`. These are abolished for you.
- Do *not* define any of the big five operations. Let the compiler handle it. This is the *rule of zero*.
- There are objects that cannot be copied, such as a stream or a mutex. If your class holds such fields, the compiler does the right thing: it forbids copying. Under certain circumstances, the ability to move may remain. Stick to the rule of zero here.

The *C++ Core Guidelines* also place great emphasis on the identification of ownership. The *Guideline Support Library* defined therein offers tools such as `owner<T>` to mark an owning raw pointer. Raw pointers without `owner<T>` do not own their target and are less problematic when copying. In [Chapter 30](#), I discuss these guidelines.

Write Your Classes Such That You Can Apply the Rule of Zero

If you use the corresponding structures of the standard library instead of the fields with ownership problems, the compiler will generate suitable copy and move operations as well as the destructor for your class.

Then you do not define or declare *any* of the big five operations for your class.

The previous example now becomes almost trivial. Depending on the use case, simply use a vector or smart pointers.

```
// https://godbolt.org/z/nbh9aozPM
#include <vector>
#include <memory>           // unique_ptr, shared_ptr
struct Type1 {              // automatic complete copy of the resource
    std::vector<char> data_;
    Type1(int n) : data_(n) {}
};
struct Type2 {              // copy prohibited, move allowed
    std::unique_ptr<int[]> data_;
    Type2(int n) : data_(new int[n]) {}
};
struct Type3 {              // copy allowed, resource is then cleanly shared
    std::shared_ptr<Type1> data_;
    Type3(int n) : data_(std::make_shared<Type1>(n)) {}
};
```

Listing 17.2 Instead of using a raw pointer, use a standard construct and do not define any of the big five operations.

17.1.4 Exceptions to the Rule of Zero

First: Third-party libraries in particular often contain things that leave incorrect ownership after copying—for example, GUI window and database handles. Often, developers do not consider harmful copies and do not prohibit them. Here, I recommend two techniques:

- Resort to manually prohibiting copying and moving. The destructor you write must then handle the correct removal, just like with a raw pointer.
- If you feel confident with C++ and smart pointers, you can use their *custom deleters* to handle the wrapped resource instead of the raw one. The smart pointer then manages the correct removal, and you can dispense with the big five. In some cases, the compiler even generates usable copy and move operations.

Second: When writing a class hierarchy that includes *virtual methods*, you must write the destructor of the base class and mark it with `virtual` (see [Chapter 15](#)). The body can be empty—and it will be, if you have followed the other guidelines of the rule of zero.

```
// https://godbolt.org/z/6vsqnWeWh
struct Base {
    virtual ~Base() {}; // define the destructor, make it virtual
    virtual void other()
};

struct Derived : public Base {
    void other() override
};

int main() {
    Base *obj = new Derived{};
    /* ... more lines of code here ... */
    delete obj;           // works because Base::~~Base is virtual
}
```

Listing 17.3 In a hierarchy with virtual methods, you must define and mark the base class destructor as virtual.

If you don't declare the destructor, the compiler will generate one. The compiler would generate it as non-virtual, and that would be bad: If the destructor is not virtual, deleting a pointer to a derived class could call the wrong destructor, as shown in the example with the `delete obj` line in `main()`. If you omit `virtual ~Base() {}`, the program would be incorrect.

But wait: the preceding program does not follow the rule of zero guidelines. I used a raw `Base *obj` pointer as an owner. That needs to be remedied. You can do this, for example, by using a `shared_ptr`. The good news is that if you use it, you won't have a problem with incorrect cleanup even if you forget the virtual destructor. The `shared_ptr` has built-in mechanisms that do more than the compiler alone: when removing its ward, it always calls the correct destructor, virtual or not.

```
int main() {
    shared_ptr<Base> obj{ new Derived{} };
    /* ... more lines of code here ... */
} // obj is correctly cleaned up
```

Listing 17.4 `shared_ptr` always calls the correct destructor, virtual or not.

That's one more reason not to use raw pointers as owners.

If you need a polymorphic pointer, use `shared_ptr` instead of raw pointers.

You will know when a pointer is *polymorphic*: when you create pointers within an object hierarchy where the classes have virtual methods—that is, when the type of the pointer variable (here `Base*`) is different from the type of the new (here `Derived*`). With `unique_ptr`, this does not work without additional mechanisms.

You can implement assignment and move assignment together by passing the argument by value (as a compiler-generated copy) and swapping the contents via `swap()`:

```
struct Data {  
  
    // ... other special functions and implementation ...  
    Data& operator=(Data copy) { // by value, assignment, move assignment  
        copy.swap(*this);  
        return *this;  
    }  
  
    void swap(Data& other) { // member function  
        using std::swap;  
        // ... element-wise swap ...  
    }  
  
    friend void swap(Data&a, Data&b); // friend: free function  
}
```

The `swap` function is useful anyway. To work well with containers, it is best to also provide the free `swap` function.

17.1.5 Rule of Zero, Rule of Three, Rule of Five, Rule of Four and a Half

There are many tips on which of the big five special functions you should implement. I will summarize the various aspects here:

- **Rule of zero**

First, check if it is possible not to implement any of the special functions. Let the compiler generate them or declare them with `= default` (this documents that you have not forgotten the aspect). This is applicable if your class does not need to manage resources itself.

- **Rule of four and a half**

If your class manages resources, starting with a destructor, implement all five—but you can combine assignment and move assignment (so, four). Use a self-implemented `swap()` (four and a half).

- **Rule of five**

Implement assignment and move assignment separately and optionally provide swap.

- **Rule of three**

Your class manages resources, but the benefit of moving is not so relevant: write a copy constructor, assignment operator, and destructor. If the compiler wants to move, it will copy instead.

17.2 Resource Acquisition Is Initialization

RAII means that when creating an object—a resource—its initialization is also done. In C++, this happens in the constructor. It is important to always leave a valid object when exiting the constructor.

This means, in particular, that even an error must not leave an invalid object lying around. If an error occurs during the initialization of the object in the constructor, you must not leave half-created data structures lying around or still blocking resources. *Resource* can mean many things here, such as a pointer to reserved memory, an open file, or a *lock* for mutual exclusion of multiple processes.

In case of an error, the constructor should be exited with an *exception* so that higher levels can also release their resources, such as requested memory. However, the exception must also be carefully triggered: previously requested resources must be released beforehand (or in a separate *catch* with *rethrow*) as the current instance does not (normally) regain control after the *throw*.

One aspect of valid is that the associated destructor must be able to completely remove the object *in any case* on its own and release all still existing resources. Taken together, this very effectively implements stack-based resource management.

17.2.1 An Example with C

A typical C API often shifts administrative tasks into the program code that uses the API. A file object or a database connection must be opened and closed again with a symmetric call. *Handles* for fonts, brushes, or audio-video codecs are obtained and explicitly released again. If the releasing call is not made—due either to an unforeseen event or simply forgetting—the associated resources are not released. The file remains open; the database connection is not closed; the handles on brushes and fonts run out.

Correspondingly, in the C world, special return values are often used to indicate an error. As a sign that the client should refrain from using the resource, many functions use specific error codes. For example, `mktime()` returns a `-1` instead of the requested conversion in case of an error. And you must check the return value of `malloc()` against the *null pointer*.

Usually, the error case in the check is the *rarely expected* result. One might almost omit the check because this case “never happens anyway.” Then one can only hope that the code is never used in a safety-critical environment.²

And as it is *rarely expected*, we come to the *exception*: in the C++ world, exceptions are available for such errors. The return value does not need to be checked because a constructor must leave an object in a *valid state*. If the constructor exits via an exception, the preallocated memory is released by the compiler. If the exception passes through multiple constructors on the *stack* (the list of current function calls) upward without being caught by a matching *catch*, then all previous constructions are undone. Only *exactly then*, when a constructor has been completely executed, is the object considered valid and fully created.

A small, lean example class that is only supposed to encapsulate a character buffer could look like the following listing.

```
// https://godbolt.org/z/9eeG575Wz
struct Buffer {
    const char *data;
    explicit Buffer(unsigned sz): data(new char[sz]) {}
    ~Buffer() { delete[] data; }
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;
};
```

Listing 17.5 An RAII character buffer class.

It should be mentioned that—true to the idea that there are “sixteen ways to stack a cat”—the same task could be better accomplished using `unique_ptr<char[]>`; this is just a simple example.

In C, this would usually involve a `malloc`, and (careful or paranoid³) programmers would need to check the return value to see if the memory was allocated correctly. If not, the process must be aborted somehow. In C++, `new` always returns a valid *pointer* or throws a `bad_alloc` exception. So there are two possible outcomes from the constructor:⁴

■ Normal run

Memory for data has been allocated, and the pointer is valid. The `buffer` object is considered correctly created, and if it needs to be cleaned up, the destructor does this.

² Warning: Irony!

³ Which are often the same thing.

⁴ Many compilers can be instructed to work entirely without exceptions. According to the standard, this is correct behavior. This is especially common in the embedded and real-time domains.

■ Exception `bad_alloc`

Memory was not allocated; the exception leaves the constructor. Thus, the `buffer` is considered not created, and no allocated resources are left behind. Because the object was not created, the destructor *will not be called* under any circumstances—even if data is uninitialized and thus a `delete[]` would be extremely dangerous. It never occurs.

The methods removed with `= delete` ensure that the `buffer` is not accidentally copied and then the entire internal data structure is freed multiple times: another reason to use `unique_ptr`.

17.2.2 Owning Raw Pointers

In complex object hierarchies, you must not forget that you also need to be prepared for the initialization of object variables (*member variables*) to fail with an exception. Even if you do not handle and swallow them with `catch`, but actually want to let the exception pass through, you may need to undo any previous initializations before the exception leaves the current constructor.

Unlike the preceding `Buffer` class, things are not so simple with `StereoImage`.

```
// https://godbolt.org/z/TK3vTv1EM
struct StereoImage {
    Image *left, *right;
    StereoImage()
    : left(new Image)
    , right(new Image) // Danger!
    { }
    ~StereoImage() {
        delete left;
        delete right;
    }
};
```

Listing 17.6 If “right” throws an exception, a leak occurs with “left”.

If the creation of `right` fails with an exception here, the memory already allocated for `left` would not be released. Because the constructor was exited with an exception, the `StereoImage` is considered not created, and its destructor will not be called. The compiler could undo the successful initialization of `left` by calling its destructor, but this is a *raw pointer* without a destructor! Another mechanism is needed—a `catch` with a *rethrow* or something other than a raw pointer for the fields of `StereoImage`.

The following listing provides one example (among many) of how you can avoid this vulnerability.

```
// https://godbolt.org/z/cs4eEEGrf
#include <memory> // unique_ptr
struct Image {
    /* ... */
};
struct StereoImage {
    std::unique_ptr<Image> left, right;
    StereoImage()
    : left(new Image)
    , right(new Image)
    { }
};
```

Listing 17.7 Correct RAI for “StereoImage”.

If the initialization of `right` fails with an exception, then `StereoImage`—as before—is considered not created. The destructor of `StereoImage` is not called. However, `left` now has a fully functional destructor, and the compiler can call it during unwinding. And that, in turn, releases the memory of `left`.

The raw pointer here occupies the resource memory. The same applies to all types of resources: *file handles*, *sockets*, *mutexes*, *semaphores*, and so on.

17.2.3 From C to C++

As noted at the beginning of the chapter, C++ developers will also frequently use C APIs. Typically, this involves *requesting* and *releasing* a resource via that API. For example, if you are using the SQLite serverless database interface, then a simplified code fragment might look like the following listing.

```
// https://godbolt.org/z/hG3j3PGYv (uses sqlite library)
#include <string>
#include <stdexcept>
#include <sqlite3.h> // library
using std::string; using std::runtime_error;

void dbExec(const string &dbname, const string &sql) {
    sqlite3 *db;
    int errCode = sqlite3_open(dbname.c_str(), &db); // acquire
    if(errCode) {
        throw runtime_error("Error opening the DB.");
    }
    errCode = sqlite3_exec(db, sql.c_str(), nullptr, nullptr, nullptr);
    if(errCode) {
        throw runtime_error("SQL Exec Error."); // ✗ not good!
```

```

    }
    errCode = sqlite3_close(db);           // release
}

```

Listing 17.8 C API in C++ code.

Here, the *database* resource is acquired with `sqlite3_open()`, released with `sqlite3_close()`, and operated on with `sqlite3_exec()` in between. It is, of course, not a good idea to exit the function with an exception:

- The first exception is correct because if `sqlite3_open()` was not successful, `sqlite3_close()` does not need to be called.⁵
- However, the second exception prevents `sqlite3_close()` from being called, and the resource would be blocked—or at least not released—which can have different impacts depending on the type of resource.

One could, of course, convert the entire SQLite3 API into a C++ API, but that is certainly a large undertaking. Besides, someone has surely already done this work for a popular API like SQLite3. Hopefully, they will also maintain it when something changes in the C API.

Rightly, one does not want to incur another dependency. A smaller solution might also suffice. A simple standard method to transform such code into RAII code is to wrap the resource in a wrapper class.

```

// https://godbolt.org/z/n7Grhh1GM
#include <sqlite3.h>
class DbWrapper {
    sqlite3 *db_;
public:
    // acquire resource
    DbWrapper(const string& dbname)
        : db_{nullptr}
    {
        const int errCode = sqlite3_open(dbname.c_str(), &db_);
        if(errCode)
            throw runtime_error("Error opening"); // prevents sqlite3_close
    }

    // release resource
    ~DbWrapper() {
        sqlite3_close(db_);           // release
    }
}

```

⁵ To be precise, the documentation states that one *should*, but not *must*. We take this example as representative of the common case in resource management, where cleanup should only occur if the request was successful.

```
// access Resource
sqlite3* operator*() { return db_; }
// No copy and assignment
DbWrapper(const DbWrapper&) = delete;
DbWrapper& operator=(const DbWrapper&) = delete;
};

void dbExec(const string &dbname, const string &sql) {
    DbWrapper db { dbname };
    const int errorCode = sqlite3_exec(*db, sql.c_str(), nullptr,
        nullptr, nullptr);
    if(errorCode)
        throw runtime_error("Error SQL-Exec."); // now it works!
}
```

Listing 17.9 C API with simple RAI.

With this wrapper, an exception can be thrown after `sqlite3_exec` without any issues. When leaving the scope of `db`—either by exception or normally—the destructor is called—thus, `sqlite3_close()`.

In the event of an exception in the constructor, the object is considered not created, and therefore the destructor with `sqlite3_close()` is not called. A good, simple RAI solution.

With the two lines with `= delete`, we prevent accidental assignments and copies that would cause multiple releases of the same resource `db_`.

17.2.4 It Doesn't Always Have to Be an Exception

Although I mainly deal with exceptions when discussing RAI in this chapter, RAI is not necessarily tied to the triggering of exceptions. You just need to recall the important principles, and the pitfalls are especially present when dealing with exceptions.

It is entirely possible to implement RAI without `throw` and to instead let users check the success of the initialization after construction—for example, with a *conversion* to `bool`.

```
// https://godbolt.org/z/c8xfM1nEa
#include <string>
#include <sqlite3.h>

class DbWrapper {
    sqlite3 *db_;
```

```

public:
    DbWrapper(const std::string& dbname)
        : db_{nullptr}
    {
        const int errCode = sqlite3_open(dbname.c_str(), &db_);
        if(errCode) db_ = nullptr; // mark as 'not successful'
    }

    sqlite3* operator*() { return db_; }

    explicit operator bool() const {
        return db_ != nullptr; // evaluate the mark
    }
    ~DbWrapper() {
        if(db_) sqlite3_close(db_);
    }
    // ... Rest as before ...
};

bool dbExec(const std::string &dbname, const std::string &sql) {
    DbWrapper db { dbname };
    if(db) { // check for successful initialization
        const int errCode = sqlite3_exec(*db, sql.c_str(), nullptr, nullptr, nullptr);
        if(errCode)
            return false; // still correct RAI
    }
    return (bool)db;
}

```

Listing 17.10 C API with simple RAI, without throw.

It is crucial that you can undo all activities of the constructor in the destructor—regardless of whether the initialization was successful or not. This is ensured here by `db_ = nullptr` and the check in the destructor. And with the help of `operator bool()`, it can also be checked from the outside whether the initialization was successful. In the example, this is also marked with `explicit`. In the standard library, for example, the *streams* work this way—but more on that later.

17.2.5 Multiple Constructors

It is also important to ensure that all ways of creating the object must leave a valid instance. In the standard library, `ostream` is an example of this:

- `ostream os` creates a data stream to a file that has not yet been specified.
- `ostream os {"file.txt"}` opens the file, if possible.

In any case, the destructor can clean up the stream again. Operations, such as via `<<`, end up in the void with the second variant if there was an error opening the file. Therefore, before using it, quickly check with `if(!os)` to see if the stream is really ready for writing. The crux of the matter here is that `os` can be assigned, opened later, and, most importantly, cleaned up correctly in any case. As an alternative design decision, an exception could have been thrown in the `ostream os{"file.txt"}` variant in case of an error. So you have to use the *C pattern*: first check the return value. This is often promptly forgotten and then leads to surprises at the next `<<`.

17.2.6 Multiphase Initialization

Sometimes, it is advisable to deviate even further from the pure RAII doctrine. Often, it is better to initialize an object in multiple phases. Typically, GUI window APIs are used in such a way that in the first step—in the constructor—the nonvisible elements are created; and then in an `init()` method, the actual visualization takes place. Between these two phases, windows and components are linked together and embedded in the layout manager, and often further dynamic tasks are performed.

But regardless of whether only the constructor and no `init` or both constructor and `init` have been executed, the destructor must always be able to completely clean up the object.

17.2.7 Define Where It Is Needed

Almost directly from RAII it follows that one should define their variables as late as possible and as early as necessary—that is, close to their actual use.

Exceptions here are, of course, tight loops, from which one might want to replace costly constructor calls by using a temporary object. However, one should not underestimate the compiler and standard library at this point; a short `string` in a tight loop might be *just barely* undesirable, but in a larger one, the `string` can be well defined where it is needed. Elementary data types, such as `int` and the like, will be optimized away by the compiler in any case.

17.2.8 “new” without Exceptions

It should not be overlooked that in some cases it can indeed be sensible and desirable to not have to consider a `bad_alloc` exception. For example, there are environments and compilers that must manage without exceptions entirely. This is common in the embedded and real-time domains, although this special treatment has decreased in recent years. And sometimes one is quite certain that no exception can be thrown here—for example, because one has written their own memory management or knows exactly that the memory request will not fail.

For this purpose, you can use a special form of `new` to enforce `nullptr` behavior.

```
// https://godbolt.org/z/s8KoE4ETs
#include <new> // nothrow
std::string *ps = new(std::nothrow) std::string{};
if(ps == nullptr) {
    std::cerr << "Memory allocation failed\n";
    return SOME_ERROR;
}
```

Listing 17.11 Nothrow-new does not throw “bad_alloc”, but returns “nullptr”.

A `new` called in this way never throws an exception (although the called constructor might). Instead, it returns a `nullptr` in the case of an error.

Tip

Prefer RAII when designing your objects: write code that utilizes RAII.

Chapter 18

Specials for Classes

Chapter Telegram

- **friend-classes**
One class can explicitly allow another class to access all its elements.
- **Signature class or pure abstract class**
In a signature class, all methods are pure virtual, i.e., virtual and = 0.
- **Multiple inheritance**
A class that directly has multiple base classes.
- **Diamond inheritance**
A class that has the same ancestor class multiple times in its hierarchy but does not include it twice. It uses virtual.
- **Literal data type**
A class that has a constexpr constructor.

Regarding classes, you now have the necessary basic knowledge. I would say you know enough for 99% of your use cases. But for those one out of a hundred classes, I also want to prepare you. This mainly involves the technical explanation for C++ regarding approaches from object-oriented programming. In C++, these missing building blocks are solved with special class forms or aspects.

18.1 Allowed to See Everything: “friend” Classes

If you have a Thing class where you have carefully hidden some things from the outside world using private or protected, but now need an exception for a specific other class Mechanic, then you can make it known within Thing.

```
// https://godbolt.org/z/sTosKfocx
#include <iostream>
class Mechanic;
class Thing {
    int value_; // private
public:
    explicit Thing(int value) : value_{value} {}
    void increment() { ++value_; }
```

```
std::ostream& print(std::ostream& os) const { return os << value_; }
friend class Mechanic;
};
class Mechanic {
    const Thing &thing_;
public:
    explicit Mechanic(const Thing &thing) : thing_{thing} {}
    auto getThingValue() const {
        return thing_.value_; // access to private member of Thing
    }
};
int main() {
    Thing thing{45};
    thing.print(std::cout) << '\n'; // output: 45
    Mechanic mechanic{thing};
    thing.increment(); // change internal value
    std::cout << mechanic.getThingValue() << '\n'; // output: 46
}
```

Listing 18.1 With “friend”, you can allow another class access to private members.

Normally, the `Mechanic` class would not be allowed access via `thing.value_`. However, because the `Thing` class considers the `Mechanic` class a “good friend” with the `friend class Mechanic` line, it is allowed to do so.

That's basically all there is to it. There are just a few more details to consider:

- **The friend class must be known**

To write `friend class Mechanic`, the name `Mechanic` must be known beforehand. For this, it is usually necessary to declare the class once with `class Mechanic`. This would not be necessary if the entire definition of `Mechanic` had already been provided upfront, but typically this is not the case.

- **Friendship applies only directly**

Friendship is not inherited. A class derived from `Mechanic` does *not* get special access to `Thing`.

And because friendship must be declared *within* the `Thing` class, no one from outside who does not have direct access to the `Thing` class can gain the special privileges of a friend class.

This fact, combined with the noninheritance of friendship, reveals one of the biggest weaknesses of this method of granting extended rights: in larger inheritance hierarchies, multiple friend classes would need to be declared. This, in turn, breaks the actual principle of encapsulation. The `friend class` already does this as it is. You should carefully consider whether your design is correct if you are tempted to use `friend class`.

18.1.1 “friend class” Example

A typical real-world example is the *abstract data structure* of a tree that stores your data at its nodes. The tree should be the interface for the user, while a node should essentially serve only as a data holder.

`unique_ptr<Node...>` is a container for a `Node` and is explained in detail in [Chapter 20](#). Because the `Tree` should hold arbitrary data, I write `Tree` and `Node` as class templates. So you will be able to write `Tree<int>`, similar to `vector<int>`. You will learn exactly how this works in [Chapter 23](#).

```
// https://godbolt.org/z/oxbcE18TM
#include <memory> // unique_ptr
#include <iostream>
#include <string>
using std::unique_ptr; using std::cout;
template <typename K, typename D> class Tree; // forward declaration
template <typename K, typename D>
class Node {
    friend class Tree<K,D>; // allow access to private members
    K key;
    D data;
    unique_ptr<Node> left, right;
public:
    Node(const K& k, const D& d) : key(k), data(d) { }
};
template <typename K, typename D>
class Tree {
public:
    void insert(const K &key, const D& data);
    D* find(const K &key) { return findRec(key, root); }
private:
    D* findRec(const K &key, unique_ptr<Node<K,D>> &node);
    unique_ptr<Node<K,D>> root;
};
template <typename K, typename D>
void Tree<K,D>::insert(const K& key, const D& data) {
    auto *current = &root;
    while(*current) { // as long as unique_ptr contains something
        auto &node = *(current->get());
        if (key < node.key) {
            current = &node.left;
        } else if (node.key < key) {
            current = &node.right;
        }
    }
}
```

```

        *current = std::make_unique<Node<K,D>>(key,data);
    };
    template <typename K, typename D>
    D* Tree<K,D>::findRec(const K& key, unique_ptr<Node<K,D>> &where) {
        if(!where)
            return nullptr;
        auto &node = *(where.get());
        if(key < node.key)
            return findRec(key, node.left);
        if(node.key < key)
            return findRec(key, node.right);
        return &node.data; // key == node.key
    };

    int main() {
        Tree<int,std::string> bt {};
        bt.insert(3, "three");
        bt.insert(2, "two");
        bt.insert(4, "four");
        auto where = bt.find(7);
        if(where==nullptr) cout<<"no 7\n";      // output: no 7
        where = bt.find(3);
        if(where!=nullptr) cout<<*where<<"\n";  // output: three
    }

```

Listing 18.2 The Tree class has access to private members of Node.

Here there is a separation: Tree is responsible for the logic, while Node holds the data. Users cannot access data in Node because key and data are both private. For Tree to read and manipulate the nodes, it is granted access to private members via friend.

I must admit, however, that the rule applies here that the use of friend class is often seen as an indication of a design flaw. It's not necessary for Node to be a user-visible class at all. It would be much better if Node were entirely in the private section of Tree. This way, the use of friend class would no longer be necessary.

“friend class” Is Something Different from “friend” for Methods

Note that a method you mark with friend is something entirely different from the friend class designation described here.

Methods you mark with friend are actually free functions and are only listed as a method within the class. They are not members of the class and do not have a this pointer. However, they retain the visibility rights of a method; see [Chapter 16](#).

A few words about the functionality of `Tree` and `Node`:

- The `Node` class holds the data of type `D` along with their keys of type `K`. In addition, two more nodes can be connected below on the right and left. I say *can* because these branches can also contain `nullptr`, indicating the end of the respective chain. Instead of raw pointers, I use `unique_ptr`, because each `Node` owns the two nodes below it and is thus responsible for cleaning them up. `unique_ptr` saves me from having to handle `delete` myself.
- The `Tree` bundles the algorithms. It holds a root node itself, under which a whole tree of `Node` elements unfolds.
- `insert` ensures that new nodes are always inserted such that for each node `n` its key with respect to the comparison is always between its two child nodes, meaning the left key is less than or equal to `n.key` and less than or equal to the right key.
- In this way, `find` or `findRec` can quickly search for `key`: Starting from `root`, where always descends the tree to the left or right, depending on whether the searched key is smaller or larger than `where.key`. Either the key is found, or `where` points to the end of the branch using `nullptr`, and the key was not found.

Binary Trees

`Tree` implements a *binary tree*. Each node can have up to two child nodes. By always branching right or left during the search, in the optimal case, each comparison can exclude half of the elements in the tree. This means that in a tree with 1,000,000 nodes, you can determine whether the key is present or not after visiting about 20 nodes.

However, note that this only applies to the *optimal* case. In the worst case, if you insert elements in sorted order, you will have to look at 1,000,000 nodes! This is because this simple algorithm does not *balance* the tree. There are sophisticated mechanisms like the *red-black tree* or *AVL tree* for this, but detailing these would go beyond the scope of this book.

If you want to store keys to data in a sorting tree structure, you should simply use `map` from Part IV. The `map` data type guarantees a balanced tree and can also handle moving and iterators.

18.2 Nonpublic Inheritance

When a class inherits something from a predecessor, you normally bring in the base class with `public`:

```
class Base {  
};  
class Derived : public Base {  
    ...  
};
```

This suggests that there are other possibilities. `protected` and `private` are also possible here, and this affects the visibility of the inherited fields and methods.

```
// https://godbolt.org/z/eKbK39nG1  
class Base {  
public:  
    int xPublic = 1;  
protected:  
    int xProtected = 2;  
private:  
    int xPrivate = 3;  
};  
class DerivedPublic : public Base {  
    // xPublic becomes 'public'  
    // xProtected becomes 'protected'  
    // xPrivate is not visible here  
};  
class DerivedProtected : protected Base {  
    // xPublic becomes 'protected'  
    // xProtected becomes 'protected'  
    // xPrivate is not visible here  
};  
class DerivedPrivate : private Base { // or if nothing is specified  
    // xPublic becomes 'private'  
    // xProtected becomes 'private'  
    // xPrivate is not visible here  
};
```

Listing 18.3 When inheriting, you can specify what should become visible.

This lists the impact of different inheritance types on member visibility, represented by the `xPublic`, `xProtected`, and `xPrivate` variables.

Although in practice `public` inheritance is almost always used, `private` is the default if you specify nothing.

For the *actual visibility* of an inherited member m in the *inheriting class*, the following rule applies when a class D inherits from a class B with visibility p :

- The actual visibility becomes p .
- But it can never be expanded.

Consider the following examples:

- For public `m` in `B` and class `D : protected B` (`p` thus protected), `m` in `D` gets the protected visibility.
- With protected `m` in `B` and class `D : private B`, `m` becomes private in `D`, because `m` inherits the visibility of `p`.
- With private `m` in `B` and class `D : protected B`, `m` remains invisible in `D`, because `m` cannot become more visible than private.
- With protected `m` in `B` and class `D : public B`, `m` is protected in `D`, because `m` cannot become more visible than protected.

One of the most obvious consequences of this rule is that a private member in the base class is always private. No visibility rule can break this. Only the class itself and friends made known within the class with `friend class` can ever access private members.

18.2.1 Impact on the Outside World

Most of this should already be clear, but the actual effects of visibility are only seen when interacting with the outside world.

```
// https://godbolt.org/z/61EdcjPn4
#include <iostream>
using std::cout; using std::ostream;
// ... as before ...
int main() {
    // public inheritance
    DerivedPublic dpu{};
    cout << dpu.xPublic << '\n';    // output: 1
    cout << dpu.xProtected << '\n'; // no access from outside
    // protected inheritance
    DerivedProtected dpt{};
    cout << dpt.xPublic << '\n';    // no access from outside
    cout << dpt.xProtected << '\n'; // no access from outside
    // private inheritance
    DerivedPrivate dpv{};
    cout << dpv.xPublic << '\n';    // no access from outside
    cout << dpv.xProtected << '\n'; // no access from outside
}
```

Listing 18.4 Inherited visibilities, viewed from the outside.

You now have no access to some data fields from the outside.

But as you know, visibility restrictions not only affect the outside: they also have implications for further derived classes.


```
// https://godbolt.org/z/Pq5aWq8xr
#include <iostream>
using std::cout; using std::ostream;
// ... as before ...
struct NormalCase : public DerivedPublic {
    void print() {
        cout << xPublic;
        cout << xProtected;
    }
};
struct SpecialCase : public DerivedPrivate {
    void print() {
        cout << xPublic;           // ✖ no access
        cout << xProtected;       // ✖ no access
    }
};
int main() {
    NormalCase n {};
    n.print();                     // Output: 12
    SpecialCase s {};
    s.print();
}
```

Listing 18.5 Inherited visibilities for further derivations.

The `SpecialCase` class inherits public from its ancestor, `DerivedPrivate`, but it has included the Base class via `DerivedPrivate : private Base`. Thus, all fields of Base became private, and further derivations like `SpecialCase` no longer have direct access.

When such an approach is chosen, the class in the middle naturally uses properties of the base class, and thus also the ultimate class, albeit indirectly.

```
// https://godbolt.org/z/WYqYe7PoY
#include <iostream>
using std::cout; using std::ostream;
class Base {
    public: int data = 5;
};
class Middle : private Base {
protected: void print() {
    cout << data;    // 'data' is inherited privately here
}
};
```

```

class Ultimately : public Middle {
    public: void go() {
        print();          // 'data' is not visible; 'print' is protected visible
    }
};
int main() {
    Ultimately u {};
    u.go();               // output: 5
}

```

Listing 18.6 In practice, child classes use privately inherited members indirectly.

In main, I instantiate `Ultimately` and call its only public method, `go`. In it, I have access to the `print()` method inherited from `Middle`. The `data` field is invisible, but indirectly, the inherited `print` method naturally uses the field.

18.2.2 Nonpublic Inheritance in Practice

In the wild, public inheritance is by far the most common. You should also follow this practice and leave protected and private inheritance for extreme special cases.

In languages like Java, all inheritance is public.

Sometimes you see private inheritance when there is not an is-a relationship but a has-a relationship. Because the interface of the inherited class `B` does not become part of the interface of the inheriting class `D`, class `D` does not appear externally as a `B` and thus is not actually a `B`.

Would you have known how to represent this has-a relationship in C++ before I explained private inheritance? Yes, you would: do not make `B` an ancestor of `D`, but rather a data member. In general, you should prefer the data member. But sometimes it is worth using inheritance. Therefore, private inheritance is sometimes also called an *implemented-by* relationship.

For protected inheritance, the same applies from an external perspective as for private inheritance. In a larger class hierarchy, there are small differences regarding visibility, but it remains an implemented-by relationship. This type of inheritance is even less useful in practice. The following listing is a concrete example of how you can implement the has-a relationship.

```

// https://godbolt.org/z/ETr73T5a9
class Engine {
public:
    Engine(int numCylinders);
    void start();           // start engine
};

```

```

class Car : private Engine {      // Car has a engine
public:
    Car() : Engine{8} { }        // initializes a car with 8 cylinders
    using Engine::start;         // starts car by starting the engine
};

```

Listing 18.7 A has-a relationship via private inheritance.

Here you see another use of `using`. There are several situations where a method from the base class is no longer visible—and this is one of them. The `start` method from `Engine` is usually only private in `Car` because `Engine` was included with private inheritance—a has-a inheritance. To make the public method visible in `Car`, you can declare the `start` identifier from `Engine` with `using Engine::start` within the appropriate section. If there are multiple overloads of `start`, all are automatically included. Alternatively, you can make it more explicit by overriding the method as `void start() { Engine::start(); }`.

Shadowing

A brief excursion: It is possible to override a method in a derived class while giving it a different signature. The overridden version is then *shadowed*—it is hidden and no longer visible:

```

// https://godbolt.org/z/Gb3zx1jf7
#include <string>
struct Base {
    void func(int x) {};
};
struct Derived : public Base {
    void func(std::string y) {}; // shadows func(int)
};
int main() {
    Derived d{};
    d.func("Text");
    d.func(3);                // ✗ Error: func(int) is no longer visible
}

```

The `d.func(3)` call can no longer see the `func(int)` overload. You would need to remove the definition of `func(string)` in the `Derived` class for it to work again (and delete `d.func("Text")` in `main()`).

Shadowing methods in this way is usually not desirable. You should assign new names. If you want to override and use virtual methods, then use `overload` to ensure you are really overriding.

The normal implementation using a member would look like the following listing. The (base) Engine class remains unchanged.

```
// https://godbolt.org/z/67EjhMf54
class Car {
public:
    Car() : engine_{8} { }           // initializes a car with 8 cylinders
    void start() { engine_.start(); } // starts the car by starting the engine
private:
    Engine engine_;                  // Car has-a engine
};
```

Listing 18.8 Has-a relationship using a member.

The commonalities of both implementations are as follows:

- In both cases, there is exactly one Engine in each Car instance.
- In neither case can users of the Car* class convert it into an Engine*, as would be the case with public inheritance.
- In both examples, Car has a start() method that leads to a call to the start() method of the contained motor.

However, there are also differences. For the inheritance variant, the following applies:

- You cannot choose this if Car is supposed to contain multiple Engine instances.
- Members of Car can convert a Car* into an Engine*.
- Methods of Car can access protected fields of Engine.
- Car could override virtual methods of Engine.

When Should You Use Private Inheritance for the Has-a Relationship?

The answer is simple: use members whenever you can, and private inheritance when you must, which is only the case when protected members or virtual methods are involved.

18.3 Signature Classes as Interfaces

Signature classes are nothing additional in C++; they don't have additional syntax requirements or keywords. They are just a name for a special form of things you already know: a class that only has pure virtual methods is a *signature class*. As a reminder, a pure virtual method is a virtual method without implementation, which is noted in the source code with = 0. Some other languages, like Java, for example, call such methods *abstract*.

A signature class is shown in the next listing.

```
// https://godbolt.org/z/rW4oTPhrG
struct Driver {
    virtual void init() = 0;
    virtual void done() = 0;
    virtual bool send(const char* data, unsigned len) = 0;
};
```

Listing 18.9 A signature class has only pure virtual methods.

This is an excellent base class for other classes like the two in the next listing.

```
// https://godbolt.org/z/cevM7nq53
#include <iostream>
using std::cout;
struct KeyboardDriver : public Driver {
    void init() override { cout << "Init Keyboard\n"; }
    void done() override { cout << "Done Keyboard\n"; }
    bool send(const char* data, unsigned int len) override {
        cout << "sending " << len << " bytes\n";
        return true;
    }
};
struct Computer {
    Driver &driver_;
    explicit Computer(Driver &driver) : driver_{driver} {
        driver_.init();
    }
    void run() {
        driver_.send("Hello", 5);
    }
    ~Computer() {
        driver_.done();
    }
    Computer(const Computer&) = delete;
};
int main() {
    KeyboardDriver keyboard {};
    Computer computer(keyboard); // Output: Init Keyboard
    computer.run();              // Output: sending 5 bytes
}                                // Output: Done Keyboard
```

Listing 18.10 A signature class makes a good base class.

For the `Computer` class, it is clear through `Driver` which methods are important. It lists the important interfaces. Distracting implementation details are not included. Therefore, this form of class is a distinct language construct in languages like Java, called an interface.

These signature classes have two main applications in C++ programs:

- When you use *multiple inheritance*, these are the only “other” classes you should inherit from. I will go into more detail on this in [Section 18.4](#).
- You can implement the runtime loading of drivers using dynamic libraries in some systems.

Let me make the example a bit more realistic. In the following listing, depending on the configuration, you can instantiate a driver for debugging or for production. Both have a signature class as a base.

```
// https://godbolt.org/z/xWr7axonx
#include <iostream>           // cout
#include <memory>             // unique_ptr
using std::cout;
class Driver {               // abstract base class
public:
    virtual void init() = 0;
    virtual void done() = 0;
    virtual void send(const std::string &data) = 0;
};
class ProductionDriver : public Driver {
public:
    void init() override { }
    void done() override { }
    void send(const std::string &data) override { cout << data << "\n"; }
};
class DebuggingDriver : public Driver {
    size_t countSend_ = 0;
public:
    void init() override {
        countSend_ = 0; cout << "Ok, I'm initialized.\n";
    }
    void done() override {
        cout << "send used:" << countSend_ << " times\n";
    }
    void send(const std::string &data) override {
        cout << "send("<<countSend_<<"):"<< data << "\n";
        ++countSend_;
    }
};
```

```
struct DriverWrapper {           // RAll wrapper for init() and done()
    Driver &driver_;
    explicit DriverWrapper(Driver& driver) : driver_(driver) {
        driver_.init(); }
    ~DriverWrapper() { driver_.done(); }
    DriverWrapper(const DriverWrapper&) = delete; // no copying
};

void doWork(Driver &driver) { // someone who flexibly uses any driver
    DriverWrapper wrapper(driver);           // call init() and done()
    driver.send("An Unexpected Journey");
    driver.send("The Desolation Of Smaug");
}

int main() {
    // same doWork, once with production and once with debugging driver
    ProductionDriver production{};
    doWork( production );
    DebuggingDriver debugging{};
    doWork( debugging );

    // more common variant of a dynamically created driver
    std::unique_ptr<Driver> driver{ new ProductionDriver{} };
    doWork( *driver );
}
```

Listing 18.11 A virtual method with implementation “= 0” is abstract.

In the `Driver` class, I made all methods `virtual`. In addition, instead of an implementation, there is `= 0`. Such a method without implementation is called *pure virtual* or *abstract*.

A class that has at least one abstract method is also called an *abstract class*. You cannot create instances of an abstract class. The compiler will respond to the following with an error:

```
Driver driver{}; // ✗ abstract class
```

You can also create large classes with many normal and virtual methods that have only one or two abstract methods, but a concrete application for abstract classes of this kind is—as the name `Driver` already suggests—that you decide during runtime whether you want to use one driver or the other. Think of a program that should work for multiple graphics cards: You create a base driver class in which all methods are abstract. All parts

of the program use references (or pointers) to this class and call the virtual methods. When your program starts, you find out whether an ATI or Nvidia graphics card is in the system and load either one or the other implementation class into memory. How this is done is extremely system dependent, and explaining it here would be out of scope of this book. However, it is feasible almost everywhere.

The basis for the type of flexibility in implementation is, in any case, that the base class *only* has abstract methods. Not a single method should be implemented. If you have another use case for the abstract class, then you can already offer as many method implementations in the abstract base class as you like.

18.4 Multiple Inheritance

It's okay if you want to experiment with multiple inheritance. But do yourself and others who read your code a favor: do not introduce multiple inheritance into long-term projects—with one exception, which is that you can use signature classes to emulate interfaces as they exist in Java.

Don't get me wrong. Multiple inheritance as a concept is fine, and it has its place in teaching object orientation. It is even necessary to model certain things well. After all, not everything in the real world can be forced into a tree-like hierarchy. In C++, however, a project becomes unmanageable if multiple inheritance is used thoughtlessly. Even if it looks good in the textbook, it is not practical to take the real world as a model. Instead, one must voluntarily restrict oneself and take technical alternatives from the C++ toolbox to be able to map the necessary world in a manageable way.¹

Do Not Use Multiple Inheritance at All, or Use It Extremely Cautiously

Stick to the rule of thumb that you should not let a class inherit implementations from two base classes.

Exception one: If you do, then do so only at the very bottom of the class hierarchy. This is useful in some programming patterns, such as the *observer*.

Exception two: As many classes without implementations (*signature classes*) can be added as you like, thus mimicking Java interfaces if you are comfortable with it.

After this warning, however, follows the explanation of what you will miss if you do not use multiple inheritance. These smaller academic examples are, of course, adequately harmless and should be clear enough.

¹ “‘Interface’ Considered Harmful,” Robert C. Martin, <http://blog.cleancoder.com/uncle-bob/2015/01/08/InterfaceConsideredHarmful.html>, 08 January 2015, [2017-02-17]


```
// https://godbolt.org/z/xnGTM351G
#include <iostream>
using std::cout;
class Mammal {
public:
    void giveBirth() { cout << "Birth!\n"; }
};
class Flying {
public:
    void fly() { cout << "Flight!\n"; }
};
class Bat: public Mammal, public Flying {
public:
    void call() { cout << "Ultrasound!\n"; }
};
int main() {
    Bat bruce{};
    bruce.giveBirth();    // Output: Birth!
    bruce.fly();          // Output: Flight!
    bruce.call();         // Output: Ultrasound!
}
```

Listing 18.12 Multiple inheritance means having multiple base classes.

The bat inherits all properties of all its base classes. For instances like `bruce`, you can call both `giveBirth()` from `Mammal` and `fly()` from `Flying`. In addition, the class can add any of its own members like `call()`.

Figure 18.1 gives a reminder of a normal class hierarchy, where each class has only one ancestor. In contrast, the `Bat` class in Figure 18.2 has multiple ancestors.

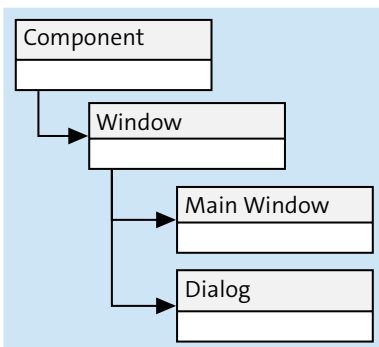


Figure 18.1 Classical inheritance.

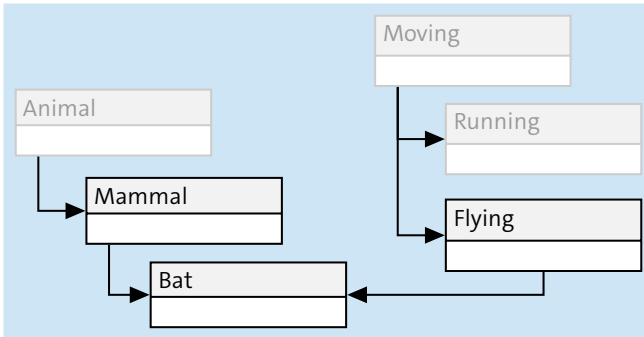


Figure 18.2 Multiple inheritance.

18.4.1 Multiple Inheritance in Practice

Although I would argue that the following example should be better implemented as a has-a relationship, it still serves as an example of an is-a relationship with multiple inheritance.

The base classes here are a clock and a calendar, combined into a clock with a calendar.² In Listing 18.13, you first see a simple clock that you can initialize, display, and advance by one second.

I show a short example here in Listing 18.13. In Chapter 15, Listing 15.3, you can see an improvement by making the example more type-safe with a little more code, especially to prevent errors when swapping constructor parameters.

```
// https://godbolt.org/z/oxoef1nTs
// Parameters must be valid values of the respective unit.
#include <iostream> // ostream
#include <iomanip>   // setw, setfill
#include <format>
using std::ostream; using std::format;
class Clock {
protected:
    int h_, m_, s_;
public:
    Clock(int hours, int minutes, int seconds)
        : h_{hours}, m_{minutes}, s_{seconds} {}
    void setClock(int hours, int minutes, int seconds) {
        h_ = hours; m_ = minutes; s_ = seconds;
    }
}
```

² “Multiple Inheritance,” Bernd Klein, http://www.python-course.eu/python3_multiple_inheritance.php

```
friend ostream& operator<<(ostream&os, const Clock& c) {
    return os << format("{:02}:{:02}:{:02}", c.h_, c.m_, c.s_);
}

void tick() { //+1 second
    if(s_ >= 59) {
        s_ = 0;
        if(m_ >= 59) {
            m_ = 0;
            if(h_ >= 23) h_ = 0;
            else { h_ += 1; }
        } else { m_ += 1; }
    } else { s_ += 1; }
}
};
```

Listing 18.13 The clock and calendar combination results in a clock with a calendar.

Using this clock is very simple:

```
int main() {
    Clock clock{ 23, 59, 59 };
    std::cout << clock << '\n'; //Output: 23:59:59
    clock.tick();
    std::cout << clock << '\n'; //Output: 00:00:00
}
```

The calendar is just as simple:

```
// https://godbolt.org/z/56Ec357Tf
// ... includes and usings ...
struct Calendar {
    int y_, m_, d_;
    static const std::vector<int> mlens_;
    bool leapyear() const {
        if(y_ % 4 != 0) return false;
        if(y_ % 100 != 0) return true;
        if(y_ % 400 != 0) return false;
        return true;
    }
public:
    Calendar(int year, int month, int day)
        : y_{year}, m_{month}, d_{day} {}
    void setCalendar(int year, int month, int day) {
        y_ = year; m_ = month; d_ = day;
    }
}
```

```

friend ostream& operator<<(ostream& os, const Calendar& c) {
    return os << format("{:04}-{:02}-{:02}", c.y_, c.m_, c.d_);
}

void advance() { // +1 day
    auto maxd = mlens_[m_-1]; // 0-based vector
    if(m_==2 && leapyear())
        maxd += 1; // February in a leap year
    if(d_ >= maxd) {
        d_ = 1;
        if(m_ >= 12) { m_ = 1; y_ += 1; }
        else { m_ += 1; }
    } else { d_ += 1; }
}

};
const std::vector<int> Calendar::mlens_ = {31,28,31,30,31,30,31,31,30,31,30,31};

```

Listing 18.14 A simple calendar class.

With `advance`, you can proceed to the next valid date. Note that here I let both the day and the date start at one (making it output-friendly), not at zero like some other APIs. Let's briefly test the calendar:

```

int main() {
    Calendar cal{ 2024, 2, 28 };
    std::cout << cal << '\n'; // Output: 2024-02-28
    cal.advance();
    std::cout << cal << '\n'; // Output: 2024-02-29
    cal.advance();
    std::cout << cal << '\n'; // Output: 2024-03-01
}

```

Now we have two practical—but separate—classes. A calendar clock now contains both.

```

// https://godbolt.org/z/zh5cPG7TK
class CalendarClock : public Clock, public Calendar {
public:
    CalendarClock(int y, int m, int d, int hh, int mm, int ss)
        : Calendar{y,m,d}, Clock{hh,mm,ss} {}
    void tick() { // +1 second
        auto prev_h = h_;
        Clock::tick(); // Call base class method
        if(h_ < prev_h) { // if new day
            advance(); // ... advance calendar
        }
    }
}

```

```
friend ostream& operator<<(ostream&os, const CalendarClock& cc) {  
    operator<<(os, (Calendar&)cc) << " "; // Call free function  
    operator<<(os, (Clock&)cc);           // Call free function  
    return os;  
}  
};
```

Listing 18.15 The calendar clock is a calendar and a clock.

And here too, I allow myself a quick test:

```
int main() {  
    CalendarClock cc{ 2024,2,29, 23,59,59 };  
    std::cout << cc << '\n'; // Output: 2024-02-29 23:59:59  
    cc.tick();  
    std::cout << cc << '\n'; // Output: 2024-03-01 00:00:00  
}
```

The `cc.tick()` call refers to the method newly defined in `CalendarClock`. However, it is also possible to call the publicly inherited method from `Calendar` with `cc.advance()`.

Did you notice that `tick()` in `CalendarClock` has overridden the `tick()` method from `Clock`? It's not so easy to access the originally overridden `tick()` anymore. To do this, you have to explicitly specify which `tick` you want by using `Clock::tick()`.

The same applies to free functions. `operator<<` is now overloaded for all three classes in our small hierarchy. In the implementation of this function for `CalendarClock`, I want to access the already existing implementation for `Calendar` and `Clock`. Here, the functions do not differ by the fact that they come from different classes: they are free functions.

The difference lies in the overloads—more specifically, in the type of the second parameter. Overall, the following overloads are declared for our classes:

```
ostream& operator<<(ostream&os, const Calendar& cc);  
ostream& operator<<(ostream&os, const Clock& cc);  
ostream& operator<<(ostream&os, const CalendarClock& cc);
```

In [Listing 18.15](#), there is a `CalendarClock&`. I get the compiler to take the other overloads by casting the parameter—that is, converting the type. As in a simple inheritance hierarchy, this also works with multiple base classes. And so, the following code sequentially calls the overloads for the two base classes:

```
operator<<(os, (Calendar&)cc) << " ";  
operator<<(os, (Clock&)cc);
```

Calling Code from Base Classes

If you explicitly need to reference a member of a base class, you can do so in different ways:

- You prefix the identifier with the base class, as in `Clock::tick()`.
- Or you cast the reference to a reference of the base class, as in `operator<<(os, (Clock&)cc)`. This also works for pointers.

This general approach is particularly useful in multiple inheritance.

18.4.2 Caution with Pointer Type Conversions

On a technical level, there is a surprise you need to be prepared for when dealing with multiple inheritance: the value of a pointer to an object can change when casting.

```
// https://godbolt.org/z/oc7jsbrEK
#include <iostream>
using std::cout;
struct Base1 {
    virtual void f1() {}
};
struct Base2 {
    virtual void f2() {}
};
struct Derived : public Base1, public Base2 {
    virtual void g() {}
};
void compare(void* a, void* b) {
    cout << (a==b ? "identical\n" : "different\n");
}
int main() {
    Derived d{};
    auto *pd = &d;
    cout << pd << '\n';           // for example 0x1000
    auto pb1 = static_cast<Base1*>(pd);
    cout << pb1 << '\n';         // still 0x1000
    auto pb2 = static_cast<Base2*>(pd);
    cout << pb2 << '\n';         // now 0x1008!
    cout << (pd==pb1 ? "same\n" : "different\n"); // Output: same
    cout << (pd==pb2 ? "same\n" : "different\n"); // Output: same
    compare(pb1, pd);             // Output: identical
    compare(pb2, pd);             // Output: different
}
```

Listing 18.16 With multiple inheritance, the value of a pointer can change.

Strange things happen here. When you print the address of `pd` with `d`, you get, for example, `0x1000`; the actual value on your machine will, of course, be quite different. So, this is the memory cell where the computer has stored the instance `d`.

If you convert this pointer to its first base class in the course of *dynamic type polymorphism*, the address remains the same: you get `0x1000` again. This conversion does not have to be as explicit as with a `static_cast`. Other possibilities include, for example, a `dynamic_cast` (where you should be prepared for more surprises anyway) or an implicit conversion when you call a function that takes a parameter of type `Base1*`.

But if you now do the same with a type conversion to the pointer of the second base class, you suddenly get a different value for the object's address—for example, `0x1008`! Huh, is the object now stored elsewhere? Well, sort of...

You usually don't notice this because when do you ever print out the value of a pointer? As you can see in the `pd==pb2` comparison, C++ *indeed* considers the two values equal—even though their values produced different outputs. This, of course, is because when comparing a `Base2*` with a `Derived*`, a conversion takes place again. Through this conversion for the comparison, the values are aligned.

But what happens if you take away C++'s chance to convert? You see this when using the `compare` function. The two parameters are of type `void*` and thus the worst thing you can do to the C++ type system. By doing this, you are telling C++: "Pointers, types, I don't care, I know what I'm doing." The type information is lost within `compare`; the direct comparison of the two pointer values now shows that they are different.

What's going on here? In C++, the members of a class (or aggregate) are stored starting from a specific address. When you derive in a simple hierarchy, you only add more members at the end; the beginning does not change. This means that all pointers to the respective beginnings of the classes within a simple hierarchy point to the same location in memory.

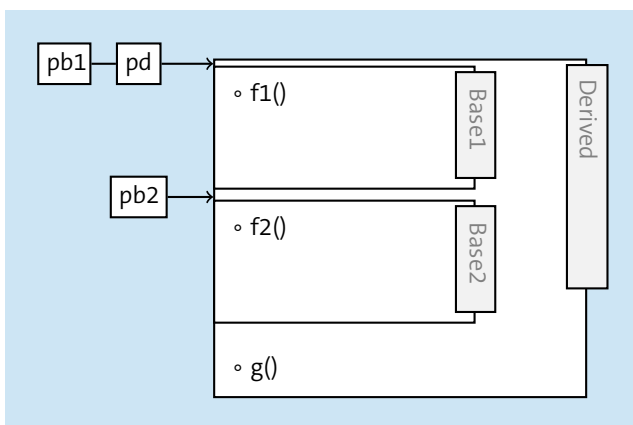


Figure 18.3 The beginning of the second base class has a different address.

This is different with multiple inheritance. The later mentioned base classes are stored in memory *after* the previously mentioned ones, as you can see in [Figure 18.3](#). The address of the Base2 subobject is therefore different from the address of the Base1 subobject or the Derived object as a whole.

When the compiler has type information, it converts pointers accordingly and adds the offsets so that you don't have to worry about it. However, if you intervene with `void*` or `reinterpret_cast` to prevent the compiler from applying its knowledge, you suppress the corrective offset of pointer values.

In all implementations I know, this address correction only happens with multiple inheritance. Theoretically, the compiler could find it necessary in other situations as well. In that case, it must guarantee that it makes the corrections for you covertly and that everything resolves itself in the case of casts and comparisons. The real culprits here would be `void*` and `reinterpret_cast`, which you should avoid anyway, right?

18.4.3 The Observer Pattern as a Practical Example

With simple inheritance, you can implement the observer pattern by adding the observer interface “laterally” and then implementing all necessary methods.

With multiple inheritance, you can inherit the implementation directly.

```
// https://godbolt.org/z/EnE7Y886M
#include <iostream>
#include <vector>
using std::cout;
// == Observer Design Pattern ==
struct Observer {
    virtual void update() = 0;
};
class Subject {
    std::vector<Observer*> observers_; // not owning pointers
protected:
    void notify() {
        for (auto o : observers_) o->update();
    }
public:
    void addObserver(Observer* o) { observers_.push_back(o); }
};
// == Concrete Class ==
struct MyThing {
    int calc() { return 1+1; }
};
```



```
// == bring together ==
struct MyObservableThing : public MyThing, public Subject {
    int calc() {
        notify();
        return MyThing::calc();
    }
};
// == observe something ==
struct MyObserver : public Observer {
    void update() override { cout << "observed\n"; }
};
int main() {
    MyObserver myObserver{};
    MyObservableThing myObservableThing{};
    myObservableThing.addObserver(&myObserver);
    auto result = myObservableThing.calc(); //Output: observed
}
```

Listing 18.17 The observer pattern with multiple inheritance.

The `Observer` and `Subject` classes form the framework of the *observer pattern*. With `MyThing`, the user class comes into play, which already does what it is supposed to do on its own. In this case, it only calculates $1+1$, but you can imagine that there is an immensely complicated fully implemented class here.

With `MyObservableThing` you now bring both things together: you get both the entire management of `Subject` on board and the actual task of the program with `MyThing`. You just need to say *what* you actually want to observe at the right place. I do this here by overriding `calc` and calling the inherited method with `MyThing::calc()`. With `notify()`, I say: “This is what the observers should notice.”

In `main`, you no longer see whether `MyObservableThing` is a multiple or simply inherited class. In any case, calling `addObserver` ensures that `MyObservable` is informed about the desired operation. Therefore, `myObservableThing.calc()` also leads to the output of `observed`.

18.5 Diamond-Shaped Multiple Inheritance: “virtual” for Class Hierarchies

[Figure 18.3](#) shows that the members of all base classes all become part of the derived class. If both base classes each have a predecessor, then this also becomes part of the final class, as illustrated in [Figure 18.4](#).

But what if the base classes of both predecessors are the same? So if `Derived1`, for example, has the `Base` base class, and `Derived2` does as well? Then, logically, `Base` is part of

both `Derived1` and `Derived2`. And if you now merge `Derived1` and `Derived2` into `DerivedDerived`, two `Bases` come together. `DerivedDerived` thus contains `Base` twice. You can see this situation illustrated in [Figure 18.5](#).

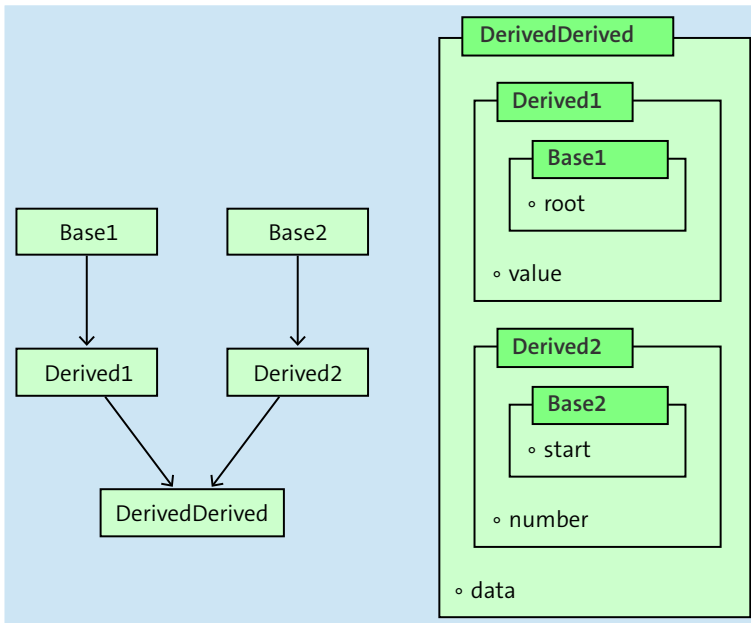


Figure 18.4 Members of the predecessors also become part of further derivations.

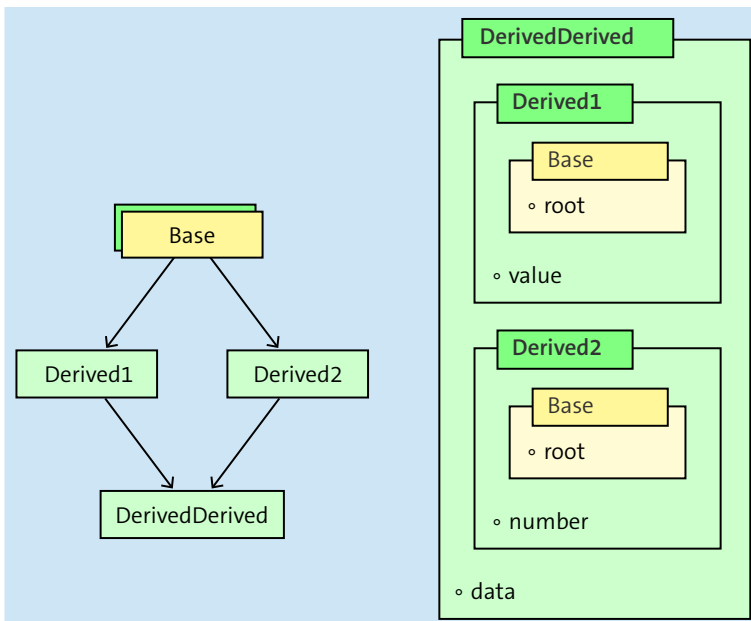


Figure 18.5 The diamond problem arises when a base class is inherited twice.

This is a problem. When a method from `DerivedDerived` wants to access the data field `root`, which one is meant? This is called the *diamond problem* or the “dreaded diamond,” and it always arises when you have a diamond-shaped inheritance hierarchy. This is an inheritance hierarchy that is not chain-like, as a simple inheritance hierarchy would be, and not tree-like, as a multiple inheritance hierarchy without duplication is, but one in which a base class of the hierarchy is mentioned as an ancestor by two derived classes.

You can explicitly state which one you want with `Derived1::root` or `Derived2::root`. Assuming this was about the color of the object being displayed, you might know that you prefer one inherited color over the other—or perhaps you want to mix both. So far, so good.

In many situations, however, this is *not* what you want. For example, if you inherit `name` as a data field and the `getName()` and `setName()` methods from both sides, you expect the object to have only one name, not two different ones.

For this purpose, there is *virtual inheritance*. The class that would appear multiple times is marked with an additional `virtual` when specifying the ancestor. This ensures that the members are not entirely within the derived class if necessary, but are stored outside—linked with a virtual connection. When multiple such base classes come together, only the virtual connection is duplicated, not the members. This is illustrated in [Figure 18.6](#).

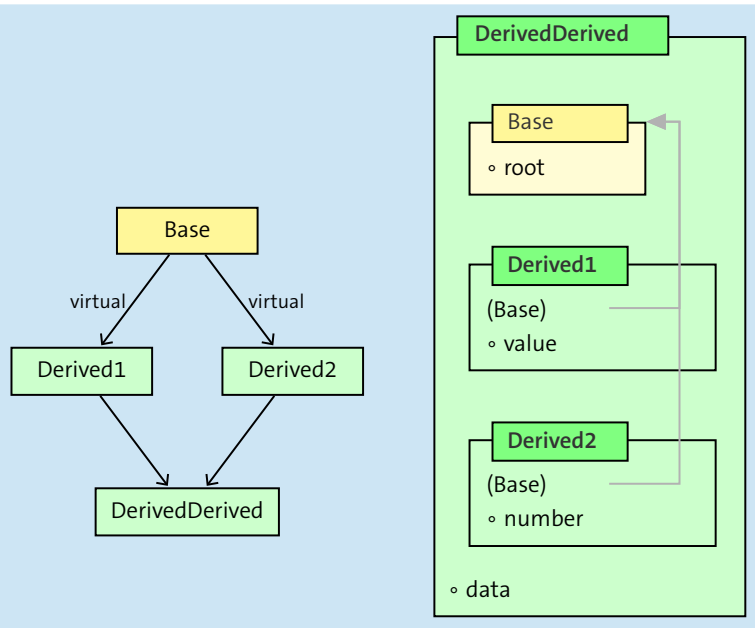


Figure 18.6 Virtual inheritance solves the diamond problem.

That was a lot of introductions. But this is also a complex topic, probably the most complex in C++ aside from templates. To give you something tangible, I will now translate the many preceding figures into code.

```
// https://godbolt.org/z/8jYeYPs7W
#include <iostream>
using std::cout;

class Base {
public:
    int data_ = 0;
};
class Derived1 : public virtual Base {
};
class Derived2 : public virtual Base {
};
class DerivedDerived : public Derived1, public Derived2 {
public:
    void method() {
        data_ = 1; // unambiguous, because there is only one data_
    }
};

void output(const DerivedDerived &dd) {
    cout << dd.data_
        << (((Derived1&)dd).data_)
        << (((Derived2&)dd).data_)
        << (((Base&)dd).data_) << '\n';
}

int main() {
    DerivedDerived dd{};
    output(dd);    // Output: 0000
    dd.method();   // sets data_ to 1
    output(dd);    // Output: 1111
}
```

Listing 18.18 A simple example with virtual inheritance.

In output, all possible levels of the hierarchy access `data_`. If there were different ones, you would address them accordingly. Here, however, `data_ = 1` in `method()` is unambiguous because there is only one `Base` in `DerivedDerived`. Without virtual inheritance, even the `(Base&)dd` cast would already be ambiguous.

You can do very fancy things with virtual multiple inheritance. You can delegate method calls to classes that are not part of the direct lineage—to sister classes, so to speak. In [Listing 18.19](#), I have written out such a case for you.

```
// https://godbolt.org/z/s9xnn7c5j
#include <iostream>
using std::cout;

struct Base {                                // abstract class
    virtual void provider() = 0;
    virtual void user() = 0;
};
struct Derived1 : public virtual Base { //still abstract
    virtual void user() override { provider(); }
};
struct Derived2 : public virtual Base { //still abstract
    virtual void provider() override { cout << "Derived2::provider!\n"; }
};
struct DerivedDerived : public Derived1, public Derived2 { //concrete
};

int main() {
    DerivedDerived dd{};
    DerivedDerived *pdd = &dd;
    Derived1* pd1 = pdd; // Cast within the hierarchy
    Derived2* pd2 = pdd; // Cast within the hierarchy
    pdd->user();           // Output: Derived2::provider()!
    pd1->user();           // Output: Derived2::provider()!
    pd2->user();           // Output: Derived2::provider()!
}
```

Listing 18.19 Effectively, `user()` calls a sister method here.

The `Base` base class defines an interface here. It is abstract because it defines *abstract methods*—that is, defined as being purely virtual with `= 0` instead of a function body. The two `Derived1` and `Derived2` classes each implement a part of this interface, but both are still abstract. Only when combined in `DerivedDerived` does the class become concrete, no longer abstract, because all methods are now defined.

Interestingly, in `Derived1`, in `user()`, the call to `provider()` results in a method being called that is actually defined only after the combination—and this by a class that is not directly connected in the inheritance hierarchy. This is called *delegating to a sister class*. This technique can be a powerful tool if you want to retroactively adjust the behavior of polymorphic classes.

18.6 Literal Data Types: “constexpr” for Constructors

You already know `constexpr`, which allows you to tell the compiler to evaluate an expression at compile time. The precomputed value is then assigned to the variable:

```
constexpr int val = 2*12+3*6;
```

In the program, the compiler will now directly insert 42 when you use `val`.

For methods and functions, `constexpr` means that, if possible, the function call is moved to compile time:

```
constexpr int add1(int v) { return v+1; }
```

This means that if you write `add1(5)` or something similar in the source code, the compiler can directly write 6 into the compiled program, and nothing needs to happen at runtime.

It is something special when you provide a *constructor* with `constexpr`. Only then can you use the data type for `constexpr` expressions and returns. This data type is then called a *literal data type*. You have already seen a short example of this in [Chapter 16, Listing 16.26](#). The following listing is another one.

```
// https://godbolt.org/z/Kvfx9YeoX
#include <array>

class Value {
    int val_;
public:
    constexpr Value(int val) : val_{val} {};
    constexpr int get() const { return val_; }
};

constexpr Value five{5}; // works, Value is a literal data type

std::array<int, five.get()> data; // works, get is constexpr
```

Listing 18.20 You create a literal data type with a `constexpr` constructor.

It is only because the constructor of `Value` is marked with `constexpr` that you are allowed to write `constexpr Value five...` at all. This, in turn, is a condition for use as a template parameter. If `five` were not `constexpr`, you could not write `array<..., five...>`. But also, the `get` must be `constexpr` here; otherwise, the compiler would not be able to get the value for the template parameter at compile time.

Literal data types are especially useful when you also define *user-defined literals* for them.

```
// https://godbolt.org/z/r1vYc7r5j
#include <array>
#include <iostream>
#include <type_traits> // is_literal_type
class Value {
    int val_;
public:
    constexpr Value(int val) : val_{val} {};
    constexpr operator int() const { return val_; }
};
namespace lit {
    constexpr Value operator"" _val(const char*, size_t sz) {
        return Value(sz); }
}
struct Nope {
    constexpr Nope() {};
    virtual ~Nope() {};           // non-constexpr destructor
};

int main() {
    using namespace lit;
    constexpr Value five{5};
    std::array<int, "11111"_val> data; // use user-defined literal
    std::cout << data.size() << '\n'; // Output: 5
    std::cout << std::boolalpha;
    std::cout << std::is_literal_type<Value>::value << '\n'; // Output: true
    std::cout << std::is_literal_type<Nope>::value << '\n'; // Output: false
}
```

Listing 18.21 User-defined literals are especially useful with literal data types.

Doesn't it seem interesting that you can use "11111"_val as a template parameter? That's because there is a continuous chain of constexpr expressions up to the return val_.

Note that you can only make particularly simple classes with a constexpr constructor into a literal data type. For example, all data fields must also be literal types. A (non-constexpr) destructor prevents such usage.

By the way, you can find out if a type is suitable as a literal data type by using the type trait `is_literal_type`. In the example, you can see this with `Nope`.

Chapter 19

Good Code, 5th Dan: Classical Object-Oriented Design

Chapter Telegram

- **Object-oriented programming (OOP)**
Object-oriented programming.
- **SOLID**
Acronym for five important OOP principles that help in the design of large projects.
- **Single responsibility principle (SRP)**
A class should have only one reason to change.
- **Open/closed principle (OCP)**
Classes, functions, and modules should be *open* for extensions but *closed* for modifications.
- **Liskov substitution principle (LSP)**
Instances of a derived type should be able to replace instances of the base type.
- **Interface segregation principle (ISP)**
No client should be forced to depend on methods it does not use.
- **Dependency inversion principle**
Modules should depend on abstractions, not on other modules.
- **Covariance**
Return types can be more specific than those of the supertype.
- **Contravariance**
Arguments can be more general than those of the supertype.
- **Model-view-controller (MVC)**
MVC is a popular architectural pattern.

In [Chapter 15](#), you have seen how to derive classes from each other in C++ using *inheritance*. In very brief terms, I also explained some central concepts of *object-oriented programming* there, particularly the difference between the *has-a* relationship and the *is-a* relationship.

However, there is more to OOP:

- **Abstraction**
You find commonalities between your objects, defining their methods and functions as generally and generically as possible.

- **Class**

A class defines the commonalities of a category of objects. This class concept is reflected in C++ in a `class`.

- **Encapsulation**

When you combine elements into a new entity, you form a capsule. So, when you define data and methods of a class, you *encapsulate*. Objects know themselves, in C++ as `this`, and manipulate only their own data directly.

- **Information hiding**

You choose which part of a class and method should be visible externally, thus forming the interface. You reveal only as much as other classes need, no more.

- **Inheritance**

A class inherits the implementation of its base class. This represents the is-a relationship.

- **Interface**

The communication interface of your program to the outside.

- **Messages**

Objects communicate with each other by sending messages to their respective interfaces. In C++, they call methods of other objects for this purpose.

- **Object**

A closed unit of data and behavior—in C++, the instance of a class. In OOP, the data of objects are often called *fields* or *attributes*. The behavior is defined in C++ using methods on the class or free functions—OOP calls these both *procedures*.

- **Polymorphism**

The ability of a program to behave differently with different objects.

- **Modularization**

More manageable units of the program through abstraction and division into classes.

The motivation for OOP is the better *reusability* of programs or modules. Abstraction should help solve problems that have already been solved with the same program code. For this to work, clean encapsulation and information hiding are necessary to minimize dependencies between program parts.

Instead of changing the individual bits and bytes of a large global state while programming, you take a step back by sending messages and look at the bigger picture. The object that receives the message is responsible for the internal details. Other objects mainly need to take care of the relationships.

By having the code more cleanly divided with fewer dependencies, the *maintainability* is also increased.

19.1 Objects in C++

In C++, OOP is represented with classes. A class is the blueprint for objects. The behavior is firmly programmed via methods, and the data is included as fields and thus as placeholders in the class definition.

When you instantiate a class, the fields are filled with concrete data, and you get a concrete object—that is, an instance of a class.

When instantiating, it doesn't matter whether you do this as an automatic variable or if it's dynamically managed:

```
VwBus bus{ green, 120_kmh};           // automatic variable
Car *modelT = new ModelT{ black, 80_kmh }; // dynamically managed
```

Both allow polymorphism. However, you must ensure that you do not accidentally copy an object according to the wrong rules. If you define a `Car` as a value parameter of a function, then you will always get a `Car`, never a `VwBus` or a `ModelT`.

```
// https://godbolt.org/z/71fE4zKhj
class Car { };
class VwBus : public Car { };
class ModelT : public Car { };
void letDrive(Car vehicle) { } // ✗ Value parameter copies only base class
void letBrake(Car &vehicle) { } // Reference parameter
void letHonk(Car *vehicle) { } // Passed as pointer
int main() {
    VwBus bus{ };           // automatic variable
    Car *modelT = new ModelT{ }; // dynamically managed
    letDrive(bus);           // ✗ gets copied to Car
    letDrive(*modelT);       // ✗ gets copied to Car
    letBrake(bus);           // remains VwBus
    letBrake(*modelT);       // remains ModelT
    letHonk(&bus);           // remains VwBus
    letHonk(modelT);         // remains ModelT
}
```

Listing 19.1 Call-by-reference as a basis for polymorphism.

Because the parameter is defined as a value parameter in `letDrive()`, C++ copies using the copy constructor. And the responsibility for *how* to copy lies with the object that is needed within `letDrive()`—and that is a `Car`. So it is the copy constructor of the `Car` class that is used here. But it only knows how to copy `Car`, not `VwBus` or `ModelT`.

However, if you pass a reference into a function like `letBrake()`, nothing needs to be copied, so the `vehicle` remains exactly the type it was outside.

Specifying the parameter as a pointer is somewhat similar to a reference. Here, only the pointer itself is the value parameter and is copied upon entering the `letHonk` function. The object itself, `bus` or `modelT`, remains the same.

“Dynamic Polymorphism” Is Not the Same as “Dynamic Objects”

In C++, you do not necessarily need *dynamic objects* (created with `new`) to achieve *dynamic polymorphism*. What is crucial is that you do not send objects as values—that is, pass them as parameters or return them.

19.2 Object-Oriented Design

Designing software projects in an object-oriented manner is difficult—even very difficult. In this book, I can only give you a little help. You will find the technical means in various other parts of the book. The methodology, on the other hand, fills other books, shelf meters worth. However, I want to give you some tips that have helped me with my designs.

When you abstract, design your classes, and form the encapsulations, you should always ask yourself about your design: “Is what I’m doing here object-oriented?” Rarely can you answer this 100% with “yes.” However, you should only abstract as much as necessary to solve your current problem cleanly.

You can find clues to see if you are still on the right track. Repeatedly questioning my design based on the *single, open, Liskov, interface, dependency* (SOLID) principles helps me, or using the SRP, OCP, LSP, ISP, or DIP acronyms.

19.2.1 SOLID

I will give an overview of these five principles before going into detail on each one:

- **Single responsibility principle**

A class should have only one responsibility and therefore only one reason to be changed. This famous rule was first formulated by Robert C. Martin (“Uncle Bob”). It specifies when you should change the *source code* of a class. He defines a reason for a source code change as a change that falls under the *responsibility* of a class—and a class should have only *one* responsibility. His example is a report whose *content* and *presentation* change. These are two responsibilities, and they should therefore be handled by two classes. Concrete implementation of this rule can be found in the *model-view-controller* (MVC) concept or the more recent *model-view-viewmodel* (MVVM) approach.

- **Open/closed principle**

Classes, functions, and modules should be *open* for extensions but *closed* for modifications. For example, a class might allow the extension of its functionality without

needing to modify its own source code. The application of abstract interfaces stems from this principle: they define a pure interface (in practice, a collection of function declarations) and implement it multiple times in different classes. This can be effectively achieved through inheritance, dynamic polymorphism, and virtual functions. In C++, you also have fewer dynamic means at your disposal, such as function overloading, template specialization, and type traits, or dynamic libraries. Coming from Java, the less dynamic approach may seem unfamiliar, but it is at least equally powerful.

- **Liskov substitution principle**

If `VwBus` is a subtype of `Car`, then you can replace objects of type `Car` with instances of type `VwBus` without altering the properties of `Car`. Because what does a `Car` need to do? Drive and honk. The derived object can also camp, but `Car` does not require this. So you can actually use `VwBus` wherever you require `Car`. For C++, this can be expressed more practically: functions that use pointers or references to a base class must, *without knowing it*, be able to handle derived classes as well.

- **Interface segregation principle**

The clean separation of interfaces ensures that a user is not dependent on methods they do not need. You avoid interfaces with many methods and split them into smaller, more specific ones, so users only need to know the methods that actually interest them. If it ever becomes necessary—for example, as part of a refactoring—you can later slice, split, and rearrange large classes differently, and only a few callers will even notice. Well-designed interfaces then help with testing because your system is less tightly coupled.

- **Dependency inversion principle**

The goal is to make higher-level modules independent of lower-level modules. You achieve this by having both sides use an abstraction layer that decouples the two module layers from each other. Here again, MVC is a good example: instead of building a GUI that is tightly coupled with the data, you separate the two into (data) model and view, then build a controller with many abstract classes in between that connects both.

Model-View-Controller

This widespread architectural pattern breaks down a program into three components that communicate in a more controlled manner than “everything with everything.” The *controller* controls the *view* (presentation) and the *model* (data model). The presentation should be independent of the way data is stored, and storing the data has nothing to do with the view being used.

For example, imagine you implement data storage in an Oracle database and the presentation for a website in HTML. The controller brings both together, and everything works.

When MVC works optimally, you could swap out the model layer so that the data is now perhaps stored in the cloud by Amazon or Google. Or you could replace the presentation with a mobile app for phones.

In theory, it all sounds clear and simple. In practice, you repeatedly encounter obstacles that try to break through this architecture. For certain problems, you have to complement the clean separation with an *observer* or take completely different approaches by slicing the components differently—as happens with MVVM.

Single Responsibility Principle

Why should a class have only a single responsibility? There are several reasons for this. The first is already embedded in the subtitle of the rule, which states that it should have only one reason to change the source code.

What does software development look like? In a larger project, different teams work on various parts of the software. And the likelihood that different people are working on different behaviors is high. If different behaviors are housed in different classes, the programmers are unlikely to get in each other's way.

On another level, program parts do not need to be installed together on the live system. With cleanly separated modules, only one module needs to be put into operation at a time. The same argumentation also simplifies testing.

Finding *the one* responsibility of a class or module is unfortunately a much more complex task than just checking off a checklist. One way to find the reasons for changes is to identify the “customers” of the class. Those who request changes in a specific area of your program or system are good candidates for these customers. Those who benefit from the class will request changes. Here are some modules and typical possible customers:

- **Persistence**

Customers include database administrators and software architects.

- **Reports**

Customers include accountants and employees in business management.

- **Payroll calculations**

Accountants, managers, and lawyers are potential customers.

- **Search module in the library catalog**

Librarians or library visitors are customers.

Assigning specific roles to concrete individuals in the software is difficult but helpful. Actual individuals can often take on multiple roles (simultaneously). Conversely, in larger systems, a role can also be filled by different physical individuals.

Finding and defining roles in the system is, in turn, difficult. As an intermediate step, the *actor* is offered. A physical person is represented by an abstract actor, who in turn is

associated with multiple concrete roles in the system. For example, Axel, the Architect would be a typical assignment of a physical person to an abstract actor, or Priscilla, the Presenter.

Now the actors are those who ask for changes in the source code. According to Robert C. Martin, it is formulated roughly like this:

- A *responsibility* is a family of functions that serves a particular actor.
- An *actor* of a responsibility is the only source for a change in that responsibility.

A classic example involves books. In the example, the books serve two roles: the first role is to be read; the second is to be found in a library.

If you try to serve both roles using a single class, it would look something like the following (pseudocode) listing.

```
// https://godbolt.org/z/fdh4n5x84
struct Book {
    auto getTitle() { return Title{"The C++ Handbook"s}; }
    auto getAuthor() { return Author{"Torsten T. Will"s}; }
    auto turnPage() { return /* Reference to the next page */ 42; }
    auto getPage() { return "current page content"; }
    auto getLocation() { return /* Shelf number/Book number */ 73; }
};
```

Listing 19.2 The Book class serves the reader and librarian actors.

Does this look “clean”? From these methods, at least two actors can be derived: the *reader*, who is mainly interested in the content of the book, and the *librarian*, who has to manage its location. It becomes difficult with the methods for title and author, as they are of interest to both actors.

If one of the two actors now demands a change in the program functionality, the change always affects the class that also concerns the other actor.

This can be solved, for example, by shifting the functions of one of the responsibilities. The following listing uses the BookFinder class for this.

```
// https://godbolt.org/z/Tq9xbGxs4
struct Book {
    auto getTitle() { return Title{"The C++ Handbook"s}; }
    auto getAuthor() { return Author{"Torsten T. Will"s}; }
    auto turnPage() { return /* Reference to the next page */ 42; }
    auto getPage() { return "current page content"; }
    auto getLocation() { return /* Shelf number/Book number */ 73; }
};

struct BookFinder {
    Catalog catalog;
```

```
auto find(Book& book) { /* Shelf number/Book number */  
    catalog.findBookBy(book.getTitle(), book.getAuthor());  
}  
};
```

Listing 19.3 The changes of an actor usually only take place in one class.

The `BookFinder` class is now the one that is interesting for the librarian. The `Book` class is the one for the reader. Of course, the `find` method can be implemented in various ways—here, for simplicity, only using title and author. However, it is important that if the requirements for library management change, this mainly concerns the `BookFinder` class; the `Book` class should not be affected. On the other hand, if a method for content summary for the reader is added, then this change probably does not affect `BookFinder`.

Open/Closed Principle

The credo of this principle is that an extension of functionality should not require a *change* to existing source code, but rather we should only write additional code that uses the existing one.

This can happen on at least two levels in modern languages like Java, C#, and C++:

- Source code that is combined by the compiler into a complete executable program
- A module dynamically loaded at runtime—in Java interchangeable `*.jar` files, in C++ dynamic libraries in the form of dynamic-link libraries (DLLs) or `*.so` files

The latter is particularly useful when it allows us to handle the program possibly even without a restart or redeployment from scratch (*redeploy*). Adding new DLLs at runtime is also conceivable, meaning that you can do so without having to completely replace any existing part.

Theoretically speaking, OCP is very simple. A simple relationship between two `User` and `Logic` classes, which are in the direct relationship “User uses Logic,” violates the OCP. If we need a second `Logic` class, from which `User` should be able to use either one or the other, there is no way around the fact that both `User` and `Logic` must be adjusted, because the connection is direct here.

This problem can be addressed with cleanly defined and separately implemented interfaces: Define a `LogicInterface` interface, which the `User` class then uses and interacts with. The interface is now concretely implemented by the two `Logic` classes, and each `User` instance can choose one of the two concrete implementations via pointer or reference.

The following example is not bad at first glance and is indeed often handled this way in practice. However, it can be easily improved in open/closed terms with inheritance and interfaces.

```

// https://godbolt.org/z/vxTsveGjT
#include <vector>
#include <memory>          // unique_ptr
#include <iostream>        // cout
enum class ShapeTag { CIRC, RECT };
struct Shape {             // Data
    ShapeTag tag_;
    double v1_, v2_;
    Shape(double w, double h) : tag_{ShapeTag::RECT}, v1_{w}, v2_{h} {}
    Shape(double r) : tag_{ShapeTag::CIRC}, v1_{r}, v2_{0} {}
};
class AreaCalculator {     // Logic
public:
    double area(const std::vector<std::unique_ptr<Shape>> &shapes) const {
        double result = 0;
        for(auto &shape : shapes) {
            switch(shape->tag_) {
                case ShapeTag::CIRC:
                    result += 3.1415 * shape->v1_ * shape->v1_;
                    break;
                case ShapeTag::RECT:
                    result += shape->v1_*shape->v2_;
                    break;
            }
        }
        return result;
    }
};
int main() {
    std::vector<std::unique_ptr<Shape>> data{};
    data.push_back(std::make_unique<Shape>(10.)); // a circle
    data.push_back(std::make_unique<Shape>(4., 6.)); // a rectangle
    // calculate
    AreaCalculator calc{};
    std::cout << calc.area( data ) << "\n";
}

```

Listing 19.4 Does not follow the open/closed principle.

The Shape “class” is a pure data class here. The data here means that either a rectangle or a circle is stored. Which one is indicated by tag_. Depending on whether only one data element for the radius is needed for a circle or two data elements for height and width are needed for a rectangle, either only v1_ or both v1_ and v2_ are relevant for the size specification.

The program or business logic is cleanly separated from the data. The `area` method in `AreaCalculator` calculates the sum of the areas of a vector. To do this, it must examine the `tag_` of each `Shape` and then decide whether to use the area formula for a circle or the formula for a rectangle.

Why “unique_ptr”?

This example would also work with `vector<Shape>` instead of `vector<unique_ptr<Shape>>`. But because I need pointers (or references) for the next example, I am introducing them here.

If the requirement arises that another type derived from `Shape`—say, `Triangle`—is added, then you must do the following:

- Add a new entry to the `ShapeTag` enumeration.
- Make the class `Shape` fit for the size specifications of a triangle.
- Extend the distinction in `AreaCalculator::area` with a new area formula.

The open/closed solution does it better:

- The objects hide their data and know their own behavior.
- The abstract base class defines an interface against which the logic is developed.

```
// https://godbolt.org/z/M6hsYa7or
#include <vector>
#include <memory>    // unique_ptr
#include <iostream>   // cout
struct Shape {
    virtual ~Shape() {}
    virtual double area() const = 0; // abstract
};
class Rectangle : public Shape {
    double w_, h_;
public:
    Rectangle(double w, double h) : w_{w}, h_{h} {}
    double area() const override { return w_ * h_; }
};
class Circle : public Shape {
    double r_;
public:
    Circle(double r) : r_{r} {}
    double area() const override { return 3.1415*r_* r_; }
};
```

```

class AreaCalculator { //Logic
public:
    double area(const std::vector<std::unique_ptr<Shape>> &shapes) const {
        double result = 0;
        for(auto &shape : shapes) {
            result += shape->area();
        }
        return result;
    }
};

int main() {
    std::vector<std::unique_ptr<Shape>> data{};
    data.push_back(std::make_unique<Circle>(10.)); // a circle
    data.push_back(std::make_unique<Rectangle>(4., 6.)); // a rectangle
    // calculate
    AreaCalculator calc{};
    std::cout << calc.area( data ) << "\n";
}

```

Listing 19.5 Follows the OCP.

The Shape class is the *interface* here, which the data classes then implement. Do you see that the AreaCalculator is implemented against exactly this interface? The area method does not take a bunch of Rectangles or Circles, but Shape.

An interface is characterized by the fact that it does not provide an implementation, but only describes function declarations. This is what Shape does by declaring all methods *pure virtual*, i.e., virtual and = 0. In other languages, this is called an *abstract method*. A class with at least one abstract method is called an *abstract class*. Abstract classes exclusively with abstract methods are called *signature classes*. This term is less known in other languages. In Java, such classes are called *interfaces*. (Java interfaces can also contain implementations with default methods, which I will ignore here.)

Dynamic Polymorphism

The area() method must not take Shape as a value parameter, because then the parameters would be copied according to the rules of Shape—that is, sliced, as shown in [Listing 19.1](#). For the same reason, vector<Shape> is also not allowed, because only sliced copies would end up in it. And because Shape is an abstract class (a class with a virtual method that is = 0), the compiler would also complain.

Therefore, vector stores pointers to Shape instances. To have cleanup done automatically, I use unique_ptr right away, but that's incidental. unique_ptr now allows *dynamic polymorphism*: each pointer knows whether it points to a Rectangle or a Circle. When I then call area(), the correct method is called *dynamically*—but only because the method is marked with virtual. You can read how this works in [Chapter 15](#).

In the example, `Rectangle` and `Circle` are each derived from `Shape`. What if some of these classes have other interfaces? You can simply add these via multiple inheritance; see [Chapter 18](#). I recommend taking Java as an example in C++ and using multiple inheritance only with signature classes; at most one ancestor should be a nonsignature class. I repeat this here because I find it so important.

OCP and SRP Are Closely Linked

Very often, a violation of SRP is also simultaneously associated with a violation of OCP and vice versa. Code that only requires a local change is likely to only result in the replacement of a module. A system that is cleanly divided with interfaces is also more likely to be cleanly divided by responsibilities.

Liskov Substitution Principle

What sounds clear and logical with vehicles like `VwBus` and `Car` is often a stumbling block in practice. This is not just about type relationships, but also about their behavior. Is an overriding method allowed to throw different exceptions? What restrictions on parameters and returns are possible (e.g., is `null` allowed)? Does a `final` method already violate the LSP if it calls any other non-`final` method? In complicated hierarchies, one often has to ask whether the LSP is being violated. Often, the matter is not so simple, and only common sense helps.

One example, or even *the* classic example, of an LSP violation revolves around the geometric shapes of the rectangle and the square. Intuitively, one might build a class hierarchy where `Rectangle` is the base class and `Square` is the derived class, because a square *is-a* rectangle, right?

```
// https://godbolt.org/z/rc67xWWfT
#include <iostream>
struct Point { int x, y; };
class Rectangle {
protected:
    Point origin_; int width_; int height_;
public:
    Rectangle(Point o, int w, int h) : origin_{o}, width_{w}, height_{h} {}
    virtual void setHeight(int height) { height_ = height; }
    virtual int getHeight() const { return height_; }
    virtual void setWidth(int width) { width_ = width; }
    virtual int getWidth() const { return width_; }
    virtual int getArea() const { return width_ * height_; }
};
class Square : public Rectangle {
public:
    Square(Point o, int wh) : Rectangle{o, wh, wh} {}
};
```

```

    void setHeight(int wh) override { width_ = height_ = wh; }
    void setWidth(int wh) override { width_ = height_ = wh; }
};
void areaCheck(Rectangle &rect) {
    rect.setWidth(5);
    rect.setHeight(4);
    auto areaValue = rect.getArea();
    if(areaValue != 20) {
        std::cout << "error!\n";
    } else {
        std::cout << "all fine\n";
    }
}
int main() {
    Rectangle rect{ {0,0}, 0,0 };
    areaCheck( rect );           // Output: all fine
    Square square{ {0,0}, 0 };
    areaCheck( square );         // Output: error!
}

```

Here I implement `Rectangle` in a natural way using origin point `origin_`, width `width_`, and height `height_`. Then isn't a `Square` just a special case of a `Rectangle`—similar to how a `VwBus` is a special case of a `Car`?

Not quite. The `Rectangle` supertype specifies that its area is calculated from height times width *and* that users can change height and width separately. You should expect that after setting height and width, the area is the product of these two dimensions.

The `Square`, which is derived from the `Rectangle`, violates this contract because it must change the width when setting the height and vice versa. So, if you set the height and width to different values one after the other, the area calculation will not return their product. Why would you set different values for height and width for a square one after the other? “That doesn't make any sense,” you might say. Not if you consider the square alone. However, you must not lose sight of the rectangle superclass. In a dynamic object-oriented world, you address the behavior of the square using the methods of the rectangle—namely, when you pass a square to functions that are also supposed to handle a rectangle.

In the example, I demonstrate this with concrete values. In `areaCheck`, I pass a `Rectangle&` as a parameter and rely on its behavior. After `setWidth(4)` and `setHeight(5)`, the area of a rectangle should be 20. Yes, for `rect`, that is the case. But if I pass a `Square&` as a parameter with `square`, this assumption is incorrect! `setHeight` simultaneously changed `width_`, resulting in a calculation of 16, not 20. That would be very confusing, given that the parameter type was `Rectangle&`.

In proper OOP, we must always ensure that we abstract real-world entities correctly. The “entities” often cannot be directly translated to “objects” one-to-one.

In LSP, there are the supertype and the subtype, for which rules apply. It does not necessarily have to be an inheritance hierarchy, but the rules are best explained in that context. Within a class hierarchy, where `Base` or `B` is the base class and `Derived` or `D` is the derived class, the following rules apply:

- **Return type can only become more specific (covariant)**

An overriding function can only impose as many or more constraints on its return type as the original function. That means, if the return type of the original function was `B`, then an overriding function can only be `B` or a `D`. If the original function is allowed to return `nullptr`, then an overriding function is also allowed to do so. The reverse is not true: if it is documented that the original function does not return `nullptr`, then no overriding function is allowed to do so. This is just one example and applies to all behavioral constraints. Throwing exceptions also falls under behavior.

- **Argument can be broadened (contravariant)**

A parameter of an overriding function can allow broader rules than the rules of the original function, but it must not narrow them. So if the original method takes type `D` as a parameter, then an overriding method can take `D`, but also `B`. If the original method does not allow `0` as a parameter value, the overriding method can allow it. Conversely, the LSP is violated if the base method allows `0` but the overriding method forbids `0`.

And what about containers? Can a base method that returns `vector<B>` be overridden by a method that returns `vector<D>`?

```
// https://godbolt.org/z/Gxh4edc8n
using std::vector;

struct B {};
struct D : public B {};

struct Base1 {
    virtual B& func();
};
struct Derived1 : public Base1 {
    virtual D& func() override;           // D& is covariant
};
struct Base2 {
    virtual B& func();
};
struct Derived2 : public Base2 {
    virtual D func() override;           // ↯ D is not covariant
};
```

```

struct Base3 {
    virtual vector<B> func();
};
struct Derived3 : public Base3 {
    virtual vector<D>& func() override; // ⚡ vector<D>& is not covariant
};
struct Base4 {
    virtual vector<B*>& func();
};
struct Derived4 : public Base4 {
    virtual vector<D*>& func() override; // ⚡ different type, not covariant
};

```

Listing 19.6 Covariance for return types.

In `Derived1`, C++ allows the `D&` return type in the overriding `func()` method. This is covariant to `B&` just as `Derived1` is to `Base1`. If the return is a value type and not a reference like `D` in `Derived2`, then this is *not covariant* because the return would be *sliced*. This can also be applied to a `vector<D>&` return type: each individual value of the container would be sliced, even if the return itself is a reference.

In the `Derived4` and `Base4` hierarchy, `vector<D*>&` is also *not covariant* with `vector<B*>&`. In C++, these are completely different types that are not related to each other. If you encounter such a case, it is acceptable to choose `vector<B*>&` as the return type of the overriding method. However, note that the LSP applies not only to the type system but also to the overall behavior.

Interface Segregation Principle

SRP is all about actors and high-level architecture. OCP deals with class design and extensibility, and LSP addresses subtypes and inheritance. ISP makes statements about the communication of components and business logic.

In all modular programs, there must be some kind of interface that users can rely on. These can be real objects that present their interface type-safely with signature classes. There are classic design patterns for this, such as facades. These are intended to regulate the user's communication with the module.

ISP is about how a module should present its interfaces to the outside.

The consideration starts with having functionality in a module that should be used by others. For example, I could identify the `Vehicle` interface with the methods from [Figure 19.1](#).

If I start this way, it will likely lead to the module being implemented in one of the following two ways:

- I could write a gigantic class that implements all the methods of the `Vehicle` interface. The sheer size of this class alone should tell you that you should not go down this path.
- Instead, I could write many small classes like `LightControl`, `SpeedControl`, and `Entertainment`, where each one implements the entire interface but only actually covers the part that is useful for itself—for example, returning *not implemented* in all unsupported methods.

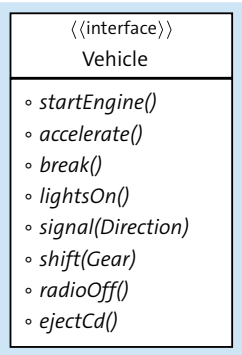


Figure 19.1 These could be the methods of a “Car” class.

It should be obvious that neither of these two options is acceptable for implementing the module.

Let's try it differently: I break the large interface into several smaller, specialized interfaces that, for example, deal with speed control or entertainment electronics, as in [Figure 19.2](#).

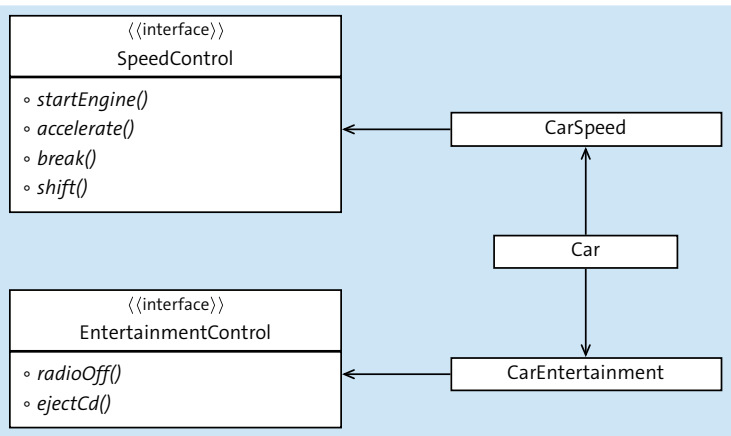


Figure 19.2 Smaller interfaces can be implemented separately.

The classes that implement the interfaces could then be used for multiple types of vehicles, such as `Car` in [Figure 19.2](#). Here, `Car` uses the `CarSpeed` and `CarEntertainment`

implementations and is dependent on the `SpeedControl` and `EntertainmentControl` interfaces through these implementations.

Actually, it is a strange perspective that `Car` relies on the `CarSpeed` and `CarEntertainment` implementations to ultimately offer the `SpeedControl` and `EntertainmentControl` interfaces. [Figure 19.3](#) illustrates the design much better.

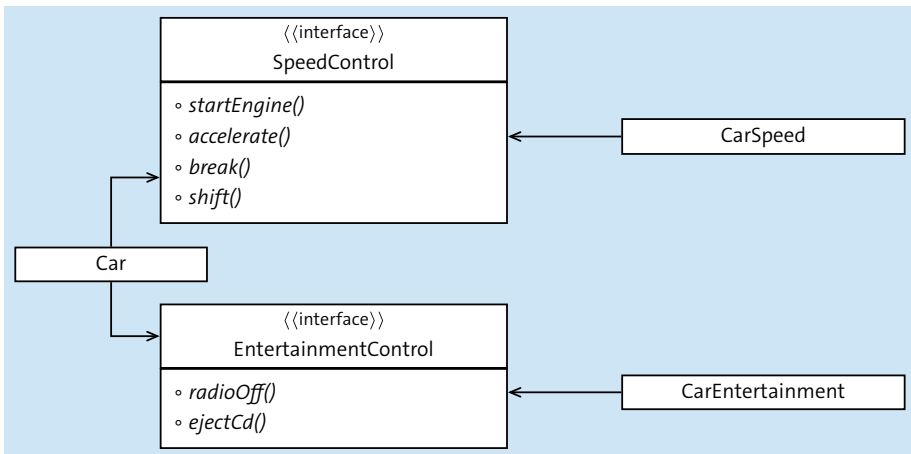


Figure 19.3 Interfaces belong to the users, not the implementations.

This fundamentally changes the perspective on architecture! `Car` becomes the *user*, instead of the *implementation*.

And despite the specialized interfaces, I want to provide users with the means to utilize the functionality of the entire module—through objects that are of type `Vehicle`. So I continue to offer the `Vehicle` interface, which unites the subinterfaces. This is the simplest solution to offer the entire functionality to other program parts like `BusStop`, `TollRoad`, and `Driver`.

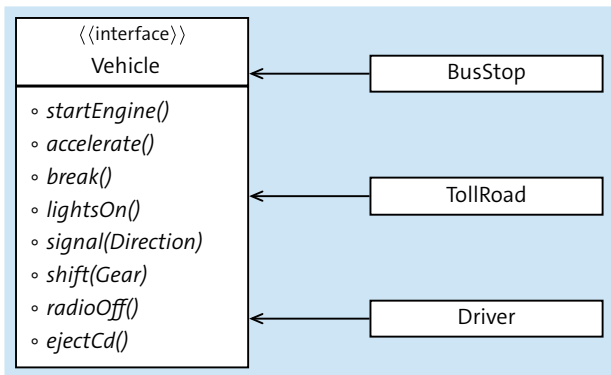


Figure 19.4 Large interfaces are not cleanly divided.

This leads to users selecting the functionalities themselves. This is definitely how most older applications and APIs are implemented.

Credo of the ISP

No client should be forced to depend on methods it does not use.

For `busStop`, `startEngine()` is an important method. But the potential danger that the interface used has become dependent on `radioOff()` violates the clean separation of interfaces.

And not only that, but it also violates SRP, as a change in `radioOff()`, for which `driver` is the user, potentially affects the `busStop`. Such a thing should never happen.

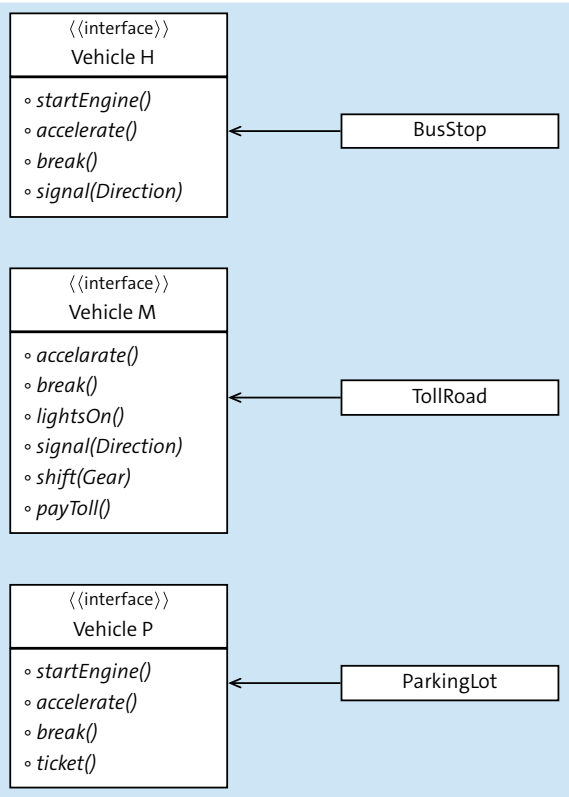


Figure 19.5 With small interfaces, users pick the appropriate ones.

Changing interfaces is always dangerous because it can have implications for the users. You usually have control over the implementation, but not always over the users. One could therefore say that interfaces belong to the users, not the implementations. Hence, the perspective from [Figure 19.3](#) is so important. You should always strive to optimize interfaces for users. Sometimes you do not yet know the actual users, which

makes it harder. In that case, it is at least good to offer several small interfaces instead of a few large ones so that users can then pick those that best meet their requirements, as shown in [Figure 19.5](#).

This leads to a certain degree of duplication. However, interfaces are just the *definitions* of the methods. The implementations within the module itself are not duplicated. The number of actual duplications is small and manageable.

To a certain extent, C++ offers you a solution through multiple inheritance. You can keep the duplicated methods in the various interfaces to a minimum by dividing them even more carefully and then expecting users to use several at the same time to a certain extent. However, this is difficult in practice for several reasons. As a reminder, multiple inheritance in this sense means that a class inherits *implementations* from several base classes. It does *not* mean that a class only inherits the *interfaces* from several base classes. On the one hand, multiple inheritance is a technical and mental challenge. On the other hand, the implementation classes within the module itself sometimes need to use multiple interfaces simultaneously to be meaningfully implemented. In that case, it is often better to offer an interface that combines the functionalities. Both the implementation and the users can then use this directly.

With ISP, it's all about keeping the needs of the users in mind. Making their needs our own ultimately improves our own code and makes our lives easier in the long run when programming.

Dependency Inversion Principle

Robert C. Martin defines this principle as follows:

- High-level modules should not depend on low-level modules. Both should depend on abstractions instead.
- Abstractions should not depend on details; details should depend on abstractions instead.

In many applications, everything eventually depends on everything else. The business logic queries the database; it constructs the data objects itself; the GUI uses the business logic, which tries to indirectly control the GUI; and so on. This is because it is the natural form of things. When I open a GUI dialog, I use data, and when the business logic says an operation is forbidden, it uses the GUI to, for example, gray out a button.

One must put in extra effort to reduce these dependencies—most of which are in the mind. One must turn the perspective upside down. Doing so, however, is extremely important. A project that implements DIP can be further developed over a longer period with reasonable effort because the connections between the components become looser and do not tangle.

The crucial point is the direction in which the dependencies point, as you can see in [Figure 19.6](#):

- The GUI or the externally offered services depend only on the business logic. Business logic is equipped with abstract interfaces, while the GUI can be concrete. Thus, the GUI becomes just a small detail of the project and not something on which other things heavily depend. This is particularly useful because the GUI tends to be quite changeable.
- The storage of objects also depends on the business logic and not the other way around. It doesn't matter which database is underlying or if even text or XML files are used. A change in the way objects are stored does not affect the business logic.
- The way instances are created is particularly useful. This also does not happen within the business logic. Making the creation independent has the advantage that it can also be changed alone. This way, simple factories are created that help generate instances. Once these are created, they perform their work within the business logic.

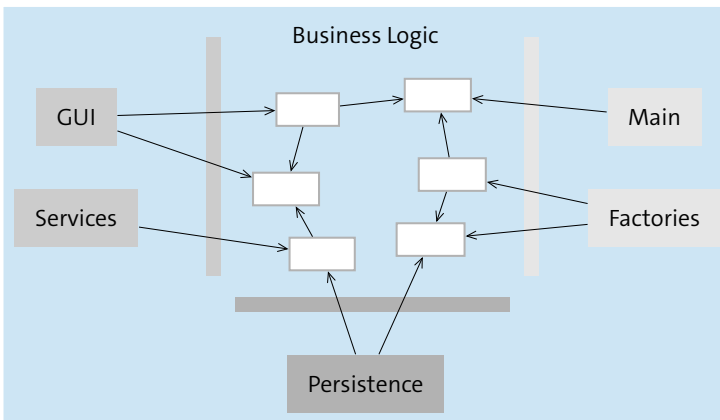


Figure 19.6 Layers do not depend on each other.

Strict adherence to the DIP has the following consequences:

- You are almost forced to comply with the OCP as well.
- You separate responsibilities and thus adhere to the SRP.
- It leads you to use inheritance properly, and you thus apply the LSP.
- It will be easier for you to separate your interfaces from the implementations. This, in turn, makes it easier to implement the ISP principle.

Thus, the circle closes. The five principles complement each other and are interdependent. The DIP holds the principles together by preventing the tangling of program code.

19.2.2 Don't Be STUPID

Sometimes it helps to mention bad examples as well. I intentionally mention these only *after* the explanation of SOLID because I find it better to always describe the

correct approach first. I will also be brief here, as the main purpose of my enumeration is to help you recognize pitfalls and know when you might be on the wrong track.

There is another acronym here. STUPID describes cases where you should pause to perhaps rethink your design:¹

- **Singleton**

A class with only one instance, often referred to as a *disguised global variable*. Singletons are controversially discussed and their use is often referred to as an *antipattern*.

- **Tight coupling**

The tight coupling of components is definitely bad and breaks several SOLID principles.

- **Untestability**

What you cannot test, you cannot change, and you cannot be sure if you are done. You can only refactor if you can test. The more automated the tests, the easier it is to change code.

- **Premature optimization**

Do not get bogged down with unnecessary details. Only deal with improvements when it pays off. Or even more extreme, if you are wondering whether to optimize, follow these rules: One, don't do it. Two (only for experts), don't do it *yet*. This is especially true for performance optimizations. In architectural matters, you will find that just writing proper tests leads to a better architecture, and the question of whether this architectural decision was the right one no longer arises.

- **Indescriptive naming**

What are `field1` to `field99`? Good names are more important than comments.

- **Duplication**

Duplicated code needs to be maintained twice and will eventually contradict itself. If you like acronyms, then stay DRY—that is, *don't repeat yourself*. When duplication threatens to arise, refactor.

As always, there are sometimes reasons for exceptions. Not every global variable and not every singleton is bad. GUIs are inherently hard to test, and some duplication simplifies the design. But exceptions should not be an excuse and should not become the rule.

¹ “From STUPID to SOLID Code!”, William Durand, 2013-08-30 <https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code>, [2024-08-12]

PART III

Advanced Topics

Part II mainly dealt with the obvious aspects of object-oriented programming, as you might know from Java or other languages. However, virtuality, inheritance, and even classes and methods are not the only way to achieve extensibility, encapsulation, and polymorphism. Arguably, you can also program in an object-oriented way in C if you put in the effort.

This part does not discuss whether other ways lead to object orientation, but it shows you possibilities beyond what has been discussed so far to design your programs and use other tools of the C++ language. You can omit elements of classical object-oriented programming and replace them with other means, achieving similar results while enjoying certain advantages.

I'll start with the bridge to the ancestor of C++, the pointers and macros of C. Through the explanation of C++ templates, we eventually come to some elements of functional programming, from which C++ has borrowed a few things to enable expressive and flexible programs.

Chapter 20

Pointers

Chapter Telegram

- **Pointer**

Variable that holds an address to data in the heap or stack.

- **Raw pointer**

Variable of type `Type *`, `Type const *` (constant value), `Type * const` (constant pointer), or `Type const * const` (both constant).

- **Heap memory or simply heap**

Place where data is stored that is requested dynamically—with `new`.

- **Stack memory or simply stack**

Memory location of objects that were created without `new`.

- **Automatic object**

An object created on the stack.

- **Dynamic object**

An object created on the heap.

- **Aliasing**

Referring to the same object from multiple variables.

- **Object ownership**

The variable that is responsible for removing a dynamic object via a pointer owns the object.

- **`new` and `new[]`**

Creating a dynamic object.

- **`delete` and `delete[]`**

Releasing a dynamically allocated object.

- **Smart pointers `unique_ptr` and `shared_ptr`**

Pointer types of the standard library following the RAII concept, which save you from explicit deallocation.

- **`move`**

Turning a named value into a temp-value, making it movable.

- **C-array**

A C-array contains multiple elements of the same type and is declared with square brackets.

- **Pointer arithmetic**

Addition, increment, and similar operations with raw pointers in a C-array.

- **C-array decay**

A C-array loses its size information when passed to a function.

- **Iterator**

The abstract concept of pointers, applied to standard containers.

Through character strings, you have already had a bit to do with pointers. You have always passed their literals around as `const char*`. And when you call `main` with arguments, the arguments are packed into a C-array of this type—that is, `const char*[]`. In this chapter, you will learn more about these constructs. They are not easy to understand but enable all sorts of useful things. However, you must use them correctly so that they do not cause you more trouble than they bring you benefits.

When you use pointers, the step to using dynamic memory is not far: `new` is the keyword here. However, I must immediately prevent a potential misunderstanding: If you use standard containers like `vector` and its relatives, then you are already using dynamic memory without realizing it. You get all the benefits but do not have to worry about the tricky details. Therefore, I would like to preface the following: in C++, you will not be able to avoid pointers in the medium term, but you should always check the alternatives first. If you even want to handle dynamic memory yourself, then do not use raw pointers for its ownership, but use the `unique_ptr` and `shared_ptr` smart pointers.

20.1 Addresses

At runtime, every “thing” must be stored somewhere. The most important things here are variables. In this sense, parameters are also variables. Other things that might not be so obvious that they need to be stored somewhere are functions and methods. In C++, functions are also just variables of a certain (complicated) type. For comparison: things that only the compiler needs and that are no longer present at runtime do not have an address, such as types themselves and the `*.cpp` files that are translated into the program. You cannot query the address of `int` or `MyType`, nor of `program.cpp`.

Every object must be stored somewhere while the program is running. For this, there is the *memory*. Technically speaking, everything that resides in memory is an *object*, and if this object has a name, it is a *variable*. You can think of memory as a street with houses, where each house can store an object. Your computer only knows this one street, so it only remembers the house number of the object. These addresses are often given in hexadecimal form. The exact form does not matter here; it also varies across different systems. For the examples in this chapter, `0xab08` and similar will represent addresses: starting with `0x`, followed by four or eight characters from `01234567890abcdef` (16 different ones—hence, *hexadecimal*).

However, for each type, there is a corresponding additional type that represents not the value itself, but the address of a value of that type. For example, if you define a variable `int value = 42;`, then `value` has the type `int`. The address of `value` has the type `int*`. You can now define a new variable of this additional type that holds the address of `value`:

```
int value = 42;
int* pValue = &value;
```

In [Figure 20.1](#), you can see how this looks in memory.

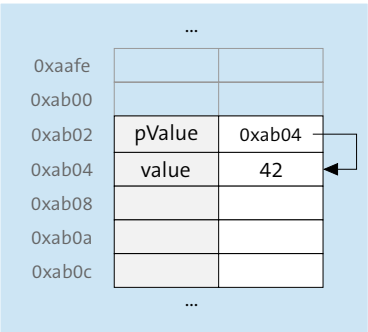


Figure 20.1 You can think of memory locations as houses on a street.

The definition of `pValue` in the second line requires an explanation. Because `pValue` is not an `int`, `int* pValue = value;` would be incorrect; you do not need the *value* from `value`, but its *address*. And this is exactly what the *unary address operator* `&` before the variable name does: `&value` gives you the address of `value`. Because `value` is of type `int`, `&value` is of type `int*`. The assignment of `&value` to `int* pValue` is therefore correct.

20.2 Pointer

The exciting thing now is that with `*pValue` you have an alternative way to access the value in `value`. When you apply the *unary dereference operator* `*` to `pValue`, it points to the memory with the 42. If you write something there, you also change the value of `value`. With `*pValue`, you have an *alias* for `value`:

```
*pValue = 18;
cout << value << "\n"; // Output: 18
```

If I may stick with the image of the street and the houses, the following happens: With `pValue` you go to its house (`0xab02`), look inside, and find the address `0xab04`. With the dereference operator `*` you follow the arrow and arrive at the house of `value`. The assignment `= 18` writes the new value to the location in memory—that is, at `value`.

This is not much different from what you could have done with references:

```
// https://godbolt.org/z/4536hxvEK
int value = 42;
int& valueRef = value; // Reference; no address operator & needed
valueRef = 18;         // no dereference operator * needed
cout << value << "\n"; // Output: 18
```

Here, the address & and dereference operator * are omitted because the valueRef name itself is an *alias* for the original variable value.

But there is one thing pointers can do that references cannot: you can assign a *new* address to a pointer.

```
// https://godbolt.org/z/7GEWYeoc9
#include <vector>
#include <iostream>
using std::vector; using std::cout; using std::ostream;
ostream& printVector(ostream& os, const vector<int> &arg) { // Helper function
    for(int w : arg) os << w << " "; return os;
}
int main() {
    vector<int> values{ };
    values.reserve(50); // Guarantee space for 50 values
    int *largest = nullptr; // Initialize with a special value
    for(int w : { 20, 2, 30, 15, 81, 104, 70, 2, }) {
        values.push_back(w);
        if(!largest || *largest < w) { // Dereference to value
            largest = &(values.back()); // Remember new address; hence not 131
        }
    }
    printVector(cout, values) << "\n"; // Output: 20 2 30 15 81 104 70 2
    // largest now contains the address of 104:
    *largest = -999; // dereference; also overwrite value
    printVector(cout, values) << "\n"; // Output: 20 2 30 15 81 -999 70 2
}
```

Listing 20.1 Pointers can be assigned new addresses during their lifetime.

First, I initialize the vector values here without elements. The pointer variable `int* largest` should always point to the largest element in the vector. Because the vector is still empty, I initialize the variable with the special `nullptr` value, which means “points nowhere.”

Then I add a few arbitrary numbers to the vector one by one. As soon as I have done that, I always check with `if(...)` whether the new value is greater than what `*largest` currently holds. Now it gets dangerous, because `largest` can contain the special value

`nullptr`, which means “points nowhere.” Dereferencing a `*nullptr` is not allowed (i.e., applying `*` or `->` to it). Therefore, I first check whether `largest` still contains `nullptr`. If so, the current value should definitely be remembered as `largest`. I could do this with `if(largest == nullptr)`. However, this can be shortened to `if(!largest)`. Together with the size comparison, this becomes `if(!largest || *largest < w)`. This means that the `largest` pointer is assigned a new value if it either still contains `nullptr` or what it points to is smaller than `w`.

If `largest` still contains `nullptr`, the expression to the right of `||` will not be executed—thanks to *short-circuit evaluation*, which is explained in [Chapter 4, Section 4.3.8](#), under the heading of the same name.

nullptr

You should never dereference a pointer that can contain `nullptr`. Check it first to see if it contains a valid address.

If the new value is larger, I get the address of the now last element with `&(values.back())` and store this as the new address value in `largest`. This happens during the program for 20, 30, 81, and at 104.

If I then write a new value to `*largest` with `*largest = -999`; between the two outputs, I actually overwrite the 104 in `vector`. You can see the result in the output.

20.3 Dangers of Aliasing

Did you notice that I secretly inserted the following into [Listing 20.1](#)?

```
values.reserve(50); // Ensure space for 50 values
```

The reason for this is that I need to avoid a danger that arises from being able to access the same value—its memory range—in different ways. That is known as *aliasing*.

The `vector` also manages its data somewhere in memory. And one of its properties is that it promises to store them consecutively. However, this means that there might be no space left at the “back” of the `vector` when you add an element with `push_back()`. But `vector` handles this: it requests a new memory range twice as large and copies (or moves) all existing values to the new area. Finally, `vector` removes the old area.

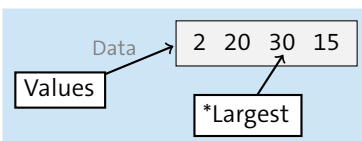


Figure 20.2 A pointer points into the data of a vector.

But in `largest` I have stored the *memory address* of a value. And if one of the `push_back()` operations triggered a vector expansion, `largest` after the expansion does not point to the actual memory location, but into the abandoned memory range. If you continue to use such a memory location, it is an error. Theoretically, anything could happen; for example, the program could crash or overwrite other data. In the present case, you might not even notice it, as you frequently retrieve the pointer anew. But if you only write smaller values after the 30, then `largest` remains at the 30. And after the automatic adjustment of the vector, you would probably just wonder why `*largest = 999`; does not affect the output afterward. The example is so short that the released memory has not yet been reused. Such errors are therefore difficult to find.

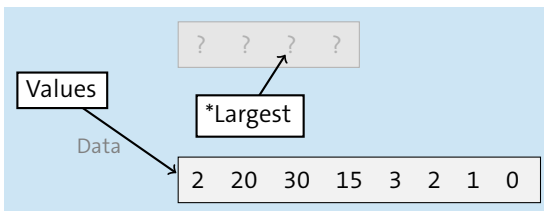


Figure 20.3 After resizing, the pointer is invalid.

With `values.reserve(50);`, I solve the specific problem in this example because vector guarantees after this call that it will not perform any resizing up to for 50 elements. Ultimately, this code is fragile: if you expand the example to more than 50 elements but forget to adjust `reserve()`, it will break. Instead, I could have used a `list` here, which does not reallocate on resizing. But this way, I was able to illustrate the aliasing problem to you. And because this is an aliasing problem and not a pointer problem, you should be aware that you could fall into the same trap with references as well.

Keep an Eye on the Lifetime of the Original When Aliasing

Whether you are working with pointers or references, you always need to consider whether the referenced object still exists. Dereferencing an invalidated object is a (frequently occurring) error.

20.4 Heap Memory and Stack Memory

20.4.1 The Stack

It's a great thing that in C++ you can rely on objects being cleaned up at a well-defined time and the associated resources being released. You know that an object created in a function is automatically removed upon exit:

```

int calculate(int param) {
    vector<int> data{ /* ... */ };
    // ...
    return result;
}

```

At the end of `calculate()`, `data` is automatically removed, no matter what you did with it in the function. The functionality is quite simple: The computer literally manages a *stack*, on which it always places the variables you define. The compiler keeps track of the positions in the stack where a range was entered—for example, with an opening block brace `{`. Upon leaving the range with the closing block brace `}`, the stack is unwound to the last marked position—see [Figure 20.4](#).

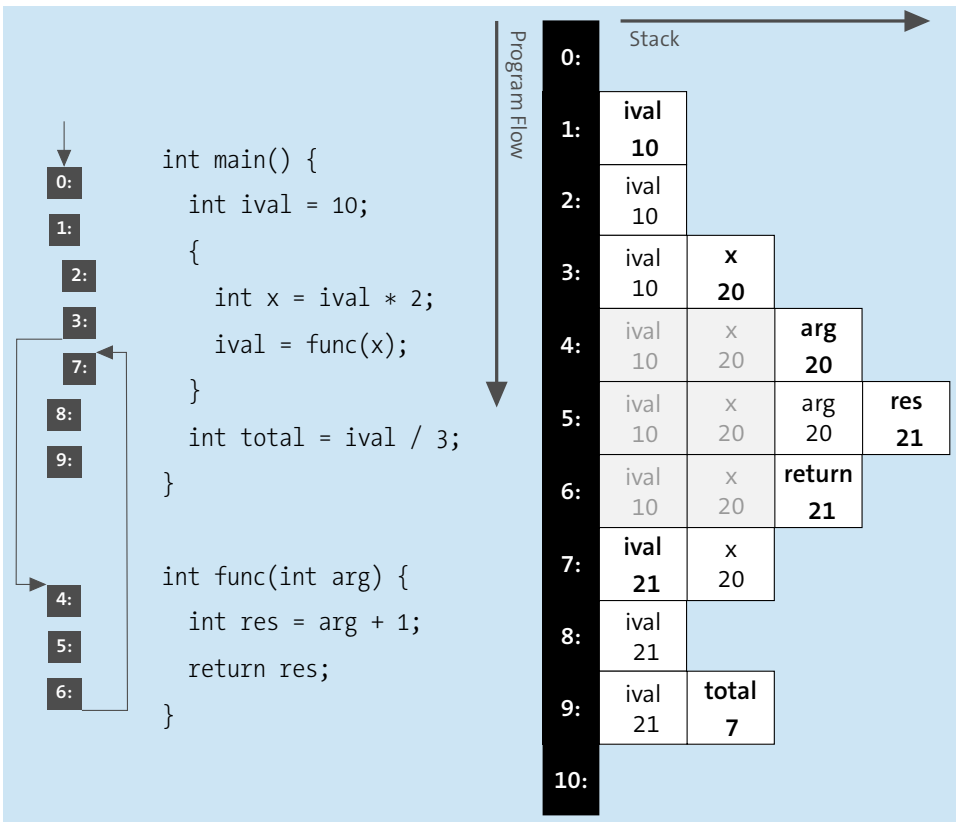


Figure 20.4 Automatic variables are managed on the stack.

This stack thus manages the *automatic variables*. Whenever a variable leaves its scope, the system removes it along with its associated resources.

In [Figure 20.4](#), you can see how variables are managed on the stack from statement to statement. Centrally, you see the executed program; between the lines, numbers are shown (on the left), which serve as timestamps for a snapshot of the stack. The stack at these respective timestamps is shown on the right side of the figure.

Upon definition, variables are placed on the stack and removed when they leave their scope. A function parameter also ends up on the stack. For example, the computer copies the `x` from `func(x)` onto the stack. The copy is then called `arg` within `func`—as seen in line 4 of the stack. Conversely, the `return` value of the function remains as the only value on the stack for the return (line 6), to then end up in `ival` with the assignment.

The gray-shaded part of the stack also indicates that `func` only has access to its own section of the stack: `ival` and `x` are not visible within the function. What is shown here for the `int` built-in type also applies to more complex data types such as standard containers and custom classes.

Multifunctional Stack

However, the stack not only contains variables, parameters, and return values. For example, the system also stores where it needs to return when it jumps to a function.

Due to some programming errors, it can happen that you write beyond the boundaries of a variable—and that means you overwrite its neighbors. While it is at least annoying if you corrupt neighboring variable values, it is deadly if you overwrite a return address. If that happens, the program will very likely crash at the next opportunity.

Or—even worse—your program becomes the target of a hacker attack and jumps into a function of a computer virus instead of its own function. It is a popular attack method to exploit programming errors in this way and inject malicious code into the computer. Therefore, it is important to always protect the stack particularly carefully from itself and others.

20.4.2 The Heap

There are cases where an object needs to live longer than the scope in which it was defined. This is also possible: you then create the object *dynamically*. This is done with the `new` keyword.

`new` itself returns a *raw pointer*, but you have already been warned about this type of pointer several times in this book. Therefore, the first thing you should do with the result of `new` is to wrap it in a *smart pointer* like `unique_ptr` or `shared_ptr`. I will cover these smart pointers later in the chapter; in this section, we will discuss what dynamic memory is.

The object created with `new` does not end up on the stack, but on the *heap*. It stays there until you explicitly remove it with `delete` (which smart pointers do automatically). For this, you need its address, which you have stored in a pointer.

However, you must not clean up a heap object twice. If you copy the pointer to the object and store it in multiple places, it is very important that you designate exactly one of the pointers as the *owner* of the object and consider all others as *users*. The owner is responsible for removing it with `delete` at the appropriate time. With raw pointers, this owner is unfortunately only in the programmers' minds and hopefully in the documentation. There are several variants of smart pointers where the ownership relationships are clear with the chosen type.

A typical use case for a heap-allocated object is when the new object needs to outlive the function in which it was created. For example, in a game framework, there might be a function that you need to override when a certain button is pressed, perhaps `createPlanet(const Event &evt)`. The return value must then be a `Planet` created within the function. As a stack object, it would be removed upon exiting the function. One solution would be to return it by value, thus as a copy, using `return`.

```
Planet createPlanet(const Event &evt) { // return by value
    Planet result{"Earth"};           // stack object
    result.setLocation(evt.getPosition());
    return result;                    // return creates (potential) copy
}
```

Listing 20.2 Returning by value can create a copy.

There is nothing wrong with this variant if it works in your project. You and the compiler can make this very performant code without a copy. Sometimes, however, returning by value is not possible or desired. In such a case, use `new` to create the object on the heap.

```
// return as (smart) pointer:
unique_ptr<Planet> createPlanet(const Event &evt) {
    unique_ptr<Planet> result{ new Planet{"Venus"} }; // heap object
    result->setLocation(evt.getPosition());
    return result; // passes the address
}
```

Listing 20.3 Returning as a pointer only copies the pointer, not the object.

You return the address of the new object wrapped in a `unique_ptr` so that you cannot forget to call `delete`.

Please excuse the use of `unique_ptr`: I wanted to present the good example first so that you remember it better than the one that works with raw pointers. Because for you, from now on, every dynamic creation of an object with `new` should always be associated with a smart pointer. Now that you know how to do it correctly, I will reduce the example to the essentials and remove `unique_ptr`. But there can be no `new` without a corresponding `delete`, so you need some code around it:


```
Planet* createPlanet(const Event &evt) {    // return as raw pointer
    Planet* result = new Planet{"Mercury"}; // heap object
    result->setLocation(evt.getPosition());
    return result;                          // passes address further
}
void userInteraction(Event &evt) {
    Planet* planet = createPlanet(evt);
    // ... do something with the planet ...
    delete planet;                          // raw pointers must be managed manually
}
```

20.5 Smart Pointers

The standard library includes a small family of pointer types that are smarter than raw pointers. They are full-fledged classes and, among other things, have a constructor that takes a raw pointer. Most importantly, they have a destructor that relieves you of the most critical task when dealing with raw pointers: calling `delete`.

The family is gathered in the `<memory>` header and includes the following members:

- **unique_ptr**
Owns the raw pointer. Not copyable, because then multiple instances would own it, but movable—for example, as a return value.
- **shared_ptr**
Owns the raw pointer, but not necessarily alone. When copied, you get two `shared_ptr` instances managing the same raw pointer; the raw pointer is only removed when the last copy is removed.
- **weak_ptr**
A little brother of `shared_ptr`, which you need in some situations when `shared_ptr` instances contain each other.
- **auto_ptr**
Deprecated since C++11; use `unique_ptr` instead. `auto_ptr` cannot be moved; if you copy it, you alter the source. This is not dangerous but is unexpected for programmers.

When I talk about smart pointers, I primarily mean the two most important representatives, `unique_ptr` and `shared_ptr`. The other two fellows work a little differently.

In addition to the constructor that takes a raw pointer and the destructor that cleans it up, the two (real) smart pointers have some similarities. Perhaps your class is similar to `Image`.

```
// https://godbolt.org/z/ehhGPjabz
#include <string>
#include <vector>
class Image {
    std::vector<char> imageData_;
public:
    explicit Image(const std::string& filename) { /* Load image */ }
    void draw() const { /* Paint image */ };
};
```

Listing 20.4 In the following examples, I use Image as a class for which I want a pointer.

With the two smart pointers, the pointer type looks as follows; you also see how to define it here:

- `unique_ptr<Image> image { new Image{"MonaLisa.jpg"} };`
- `shared_ptr<Image> image { new Image{"TheScream.jpg"} };`

However, it is cumbersome to have to write Image twice. The feature added in C++17 that allows `<Image>` to be deduced from the constructor argument does not help here because it is an `Image*`. Therefore, there are helper functions for creation:

- `auto image = make_unique<Image>("MonaLisa.jpg");`
- `auto image = make_shared<Image>("TheScream.jpg");`

`make_unique()` was only added in C++14, but if your compiler supports `make_shared()` from C++11, it probably also comes with `make_unique()`.

When you have the instance of the smart pointer in your hands with `image`, you can use these methods:

- The `get()` method returns the wrapped raw pointer to you. This is useful when you need to pass the pointer to a function that takes raw pointers as arguments. In that case, the function is only a *user*, but not an *owner*.
- The unary operator `*` works like `get()`, followed by a `*` dereference. So you can write either `*(image.get())` or simply `*image`. In both cases, you get `Image&`.
- With operator `->`, you can directly access a member of the wrapped object, such as `image->draw()`.
- With `reset()`, you can prematurely delete the raw pointer and trigger the proper release of the managed object using `delete`. After `image.reset()`, `image.get()` returns `nullptr`. Optionally, you can also pass another raw pointer as an argument, which the smart pointer will then take ownership of.
- With `swap(a,b)`, you can swap contents. This is important when copying is forbidden or expensive.

Use “make_unique” and “make_shared”

You should get into the habit of preferring the helper functions for constructing smart pointers. If you use the constructor and pass the result to a function, memory leaks can occur:

```
// takes two arguments, at least one smart pointer:
func(shared_ptr<Image> a, shared_ptr<Image> b);
int main() {
    func(new Image{"a.jpg"}, new Image{"b.jpg"}); // dangerous code
    func(make_shared<Image>("a.jpg"), make_shared<Image>("b.jpg")); // clean
}
```

20.5.1 “unique_ptr”

`unique_ptr` is the first choice when there should be a single owner of the pointer at any given time—that is, someone who will clean up the wrapped raw pointer at the end. There are mainly two use cases:

- `unique_ptr` never leaves the scope in which it was defined, and thus the raw pointer associated with it is cleaned up when `unique_ptr` ends its lifetime. This applies to all `unique_ptr`s—for example, as a local variable or as a data member of a class.
- Or there is a unique path from a source to a sink. For example, `unique_ptr` can be the return value of a function. The compiler transfers the content—the raw pointer—from the inner `unique_ptr` to the outer one.

```
// https://godbolt.org/z/aTTejrWE
#include <memory> // unique_ptr
#include <string>
#include <iostream>
using std::unique_ptr; using std::string;
class Component { }; // Dummy window hierarchy
class Label : public Component { };
class Textfield : public Component { };
class Button : public Component {
public:
    int id_; // ID to distinguish the buttons
    explicit Button(int id) : id_{id} {}
};
class Window { };
class MyDialog : public Window {
    string title_;
    unique_ptr<Label> lblFirstName_{new Label{}}; // lots of data fields
    unique_ptr<Textfield> txtFirstName_{new Textfield{}}; // ... tied to the lifetime
};
```

```

    unique_ptr<Label> lblLastName_{new Label{}};           // ... of the class
    unique_ptr<Textfield> txtLastName_{new Textfield{}};
    unique_ptr<Button> btnOk_{new Button{1}};
    unique_ptr<Button> btnCancel_{new Button{2}};
public:
    explicit MyDialog(const string& title) : title_{title} {}
    unique_ptr<Button> showModal()
        { return std::move(btnOk_); }    // Placeholder code; OK pressed
};
unique_ptr<MyDialog> createDialog() {
    return unique_ptr<MyDialog>{ // temporary value
        new MyDialog{"Please enter name"}};
}
int showDialog() {
    unique_ptr<MyDialog> dialog = createDialog();        // local variable
    unique_ptr<Button> pressed = dialog->showModal();    // return value
    return pressed->id_;
}
int main() {
    int pressed_id = showDialog();
    if(pressed_id == 1) {
        std::cout << "Thank you for pressing OK\n";
    }
}

```

Listing 20.5 `unique_ptr` as data field, return value, and local variable.

In Listing 20.5, I have extremely simplified a window application that displays a dialog. Users are supposed to press one of two buttons, and the program returns the pressed button as a number—but we consider windows schematically here, and in the brevity of the program no actual windows pop up. However, you will find a similar class hierarchy in most window programming interfaces.

Because all `unique_ptr` data fields belong to the `MyDialog` class (the class *owns* them), they are correctly removed when `dialog` is destroyed in `showDialog()`, along with all `unique_ptr`s. These, in turn, own their respective raw pointers and clean up their charges with `delete` in their destructor.

Something interesting happens in `createDialog()`. The `return` expression creates a new value—even a temp-value, essentially a variable without a name. Does this mean that at the end of the `return` statement, `unique_ptr<MyDialog>{new MyDialog{"..."}}` is immediately cleaned up again? Yes, it is, but not without first being “copied” for return to end up in the variable `dialog` in `showDialog()`. Did I say copied? Oh, no, that’s not possible! You cannot copy a `unique_ptr`. If you could copy it, there would be two `unique_ptr` instances, both wanting to manage the same raw pointer. That would not be “unique.”

Therefore, `unique_ptr` is not copied but *moved*. To be precise, its content is moved. This means that `unique_ptr` transfers the raw pointer managed by the newly created temp-value (inside) into the `unique_ptr<MyDialog> dialog` (outside).

In `showModal()`, it gets even more interesting. In principle, the same thing happens here as in `createDialog()`. But this method returns an *existing* value by moving it. But why doesn't it just say `return btnOk_`? That's because `btnOk_` is not a temp-value. As you saw in [Chapter 16](#), you can safely “steal” only from objects that you know will soon disappear. This is the case with the temporary value. However, `btnOk_` is not a temporary value but a data field—an object with a name. If the compiler were to silently take its raw pointer, it would be a surprise.¹

Well, here you are sure that you want to extract the content from the existing variable—move it as if the variable were a temp-value. With `std::move`, you tell the compiler: “Yes, dear compiler, please consider `btnOk_` as a temp-value,” and lo and behold, it behaves like in `createDialog()`. That is, `unique_ptr` takes the raw pointer from the return value and transfers it to pressed in `showDialog()`.

So far, so good: the program does what it should, and you see the text output, “Thank you for pressing OK”. What you must not do, however, is call `dialog->showModal()` twice:

```
int showDialogAgain() {
    unique_ptr<MyDialog> dialog = createDialog();
    unique_ptr<Button> pressedOne = dialog->showModal();
    unique_ptr<Button> pressedTwo = dialog->showModal(); // ⚡ not twice
    return pressedTwo->id_; // ⚡ Error; likely crash
}
```

You have surely recognized that by calling `showModal()` for `pressedOne`, the `btnOk_` data field from `dialog` is transferred to the `pressedOne` variable. Both are `unique_ptr`, but only one can be the owner of the raw pointer: the `Button*` created once with `new`. And with the first call to `showModal()`, `pressedOne` now contains this `Button` pointer, and `dialog.btnOk_` contains `nullptr`, the value for “empty.” In this example, the second `showModal()` still works; you then get this `nullptr` transferred to `pressedTwo`. If you then try `->id_` on this `nullptr`, the program will (at best) crash.

Excuse this little tutorial on what you can do with `unique_ptr` as a return value. It illustrates that `unique_ptr`, with its unique ownership, has side effects. These are exactly the side effects you want: the compiler would not come up with the idea of stealing the contents from a variable or a data field with a name on its own. I helped with `std::move()`.

¹ Note that this is exactly what `auto_ptr` would do. Therefore, please do not use it from C++11 onward.

“std::move” Itself Does Not Move

Because you are seeing `std::move()` here for the first time, I want to give a little hint, without which you might fall into a misunderstanding. The `std::move()` function does *not* move itself, it only *allows* something to move—here, the return from the function.

Really this is just a type conversion to a temp-value reference, and some would have preferred a name like `move_cast`.

The data fields are all `unique_ptr` pointers. Simple data fields without pointers would have sufficed here. But maybe you want to extend `MyDialog` polymorphically later: for that, you need pointers. *Polymorphic* here means that you, for example, create a derived `ColorfulTextfield` class from class `Textfield` and then put an instance of that class into `unique_ptr<Textfield> txtFirstName_`. If `txtFirstName_` were not a pointer (or a reference), the properties of `ColorfulTextfield` would be lost. You can find an example of this in [Chapter 15](#).

You can apply the rules of thumb for using `unique_ptr`. Exceptions prove the rule:

- You can most likely use `unique_ptr` as a reference parameter. Then the parameter behaves like any other reference. In addition, you can also pass `nullptr` for “undefined state,” which is sometimes useful.
- A value parameter as `unique_ptr` “consumes” the parameter upon invocation. The variable that was passed to the function is empty afterward. This is useful when the object should take exactly one path. This would be the application of the *sink* (or *drain*) *design pattern*.
- A function can return `unique_ptr` by value. This is particularly suitable when you create the wrapped object with `new` within the function—essentially as a *producer function* or *factory*.
- You only return a reference to `unique_ptr` if the pointer already existed beforehand. So, from a free function, you return a parameter, or from a method, a data field of the class.
- As data fields of a class, `unique_ptr`s are well-suited because you don't have to worry about removal. Unlike a value, `unique_ptr` can have an undefined phase and store `nullptr`. It is certainly better than a raw pointer.
- You can put `unique_ptr` in a standard container. The container then owns all the objects it contains. If you need to reference individual container elements from another location, you can do so either by reference to `unique_ptr` or as a raw pointer to the wrapped object. The latter is useful if you don't want to reference multiple elements of the original container in another container that contains the objects.

20.5.2 “shared_ptr”

The `shared_ptr` is also for regulated ownership, but it is not quite as possessive. `shared_ptr` is willing to share the object it owns with many others—so long as they are also `shared_ptr` or `weak_ptr`. The wrapped object exists only once. When the last `shared_ptr` that references this object disappears, the object also disappears.

You can use this mechanism when it is not clear who owns the objects and when exactly they should be cleaned up—or even when you know for sure that there are times when multiple entities own the objects. All in all, there are probably many more different use cases for `shared_ptr`, precisely because it allows so much freedom in terms of ownership.

```
// https://godbolt.org/z/4KP91z1bq
#include <vector>
#include <iostream>
#include <memory> // shared_ptr
#include <random> // uniform_int_distribution, random_device
namespace { // Beginning of the anonymous namespace
using std::shared_ptr; using std::make_shared;
using std::vector; using std::cout;
struct Asteroid {
    int points_ = 100;
    int structure_ = 10;
};
struct Ship {
    shared_ptr<Asteroid> firedLastOn_{};
    int score_ = 0;
    int firepower = 1;
    bool fireUpon(shared_ptr<Asteroid> a);
};
struct GameBoard {
    vector<shared_ptr<Asteroid>> asteroids_;
    explicit GameBoard(int nAsteroids);
    bool shipFires(Ship& ship);
};
// Implementation of Ship
bool Ship::fireUpon(shared_ptr<Asteroid> a) {
    if(!a) return false; // invalid asteroid
    a->structure_ -= firepower;
    if(a.get() == firedLastOn_.get())
        firepower *= 2; // increase damage
    else
        firepower = 1; // reset
    firedLastOn_ = a;
}
```

```

        return a->structure_ <= 0;          // destroyed?
    }
// Implementation of GameBoard
GameBoard::GameBoard(int nAsteroids) : asteroids_{}
{
    // some standard asteroids
    for(int idx=0; idx<nAsteroids; ++idx)
        asteroids_.push_back( make_shared<Asteroid>() );
}
int rollDice(int min, int max) {
    /* static std::default_random_engine e{}; */ // Pseudo-random generator
    static std::random_device e{}; // random generator
    return std::uniform_int_distribution<int>{min, max}(e); // roll dice
}
bool GameBoard::shipFires(Ship &ship) {
    int idx = rollDice(0, asteroids_.size()-1);
    bool broken = ship.fireUpon(asteroids_[idx]);
    if(broken) {
        ship.score_ += asteroids_[idx]->points_;
        asteroids_.erase(asteroids_.begin()+idx); // remove
    }
    return asteroids_.size() == 0;                // everything is destroyed
}
} // End of the anonymous namespace
int main() {
    GameBoard game{10};                          // 10 asteroids
    Ship ship{};
    for(int idx = 0; idx < 85; ++idx) {           // 85 shots
        if(game.shipFires(ship)) {
            cout << "The space is empty after " << idx+1 << " shots. ";
            break;
        }
    }
    cout << "You have scored " << ship.score_ << " points.\n";
}

```

Listing 20.6 A game board with various objects.

If you run this program, you might be able to clear the “space” with some luck. Here are the results of a few runs:

```

You have reached 700 points.
You have reached 800 points.
You have reached 800 points.
You have reached 600 points.
Space is empty after 84 shots. You have reached 1000 points.

```


You have reached 700 points.
You have reached 800 points.
Space is empty after 82 shots. You have reached 1000 points.
You have reached 800 points.
You have reached 900 points.

As the program is set in `main()`, 10 asteroids are generated as targets on the game board. Each starts with ten “structure” points; when these reach zero, the asteroid is destroyed. The last line of `Ship::fireUpon()` checks this. This method is called by the main method of the `GameBoard::shipFires()` minigame. Because the game board manages the asteroids as data fields, it is responsible for removing a destroyed asteroid from the “universe.” This happens after the line with `ship_.fireUpon()`: the method itself reports with the return value whether the asteroid was destroyed. If that is the case, the ship gets points, and the hit rock is removed from the vector with `asteroids_.erase()`.

Which of the available asteroids the ship shoots at is determined by a die roll. For this, there is the `rollDice()` helper function. With the two `min` and `max` parameters, you specify how many sides the die should have and what the smallest rolled number is. The die roll itself is done using `std::uniform_int_distribution<int>{min, max}(e)`. If you were to pass 1 and 6 as `min` and `max`, you would have a six-sided die. Because `max` is always adjusted to the number of available asteroids, this “die” always has a different number of sides. Note that with `random_device`, you get true random numbers. However, because debugging with true random numbers is difficult, you can also use `default_random_engine` instead. Then you always get the same sequence of numbers that only *looks* random. This is useful for debugging.

Above all, you can see in this example that `GameBoard` stores all the asteroids in a `vector<shared_ptr>`. I chose `shared_ptr` here because a `Ship` should also point to an `Asteroid` in this vector. Thus, both “own” an asteroid. If the `GameBoard` removes an asteroid from its vector with `erase`, the `Ship` still has “its” last targeted asteroid in sight and can query information about it; this can be useful in a more complex game. However, if `Ship ship{}` targets the next asteroid in the next dice round, the ship's `shared_ptr` relinquishes ownership of the previous one. Either it is now removed if the vector of the `GameBoard` no longer owns it, or it remains if it is still present. All in all, this is very convenient: you only have to worry about very little and don't need to be concerned.

Due to these automatisms, it is sometimes said that `shared_ptr` operates like Java *garbage collection*, but this is not the case. Java circumvents the problem of unique ownership through garbage collection. In C++, however, `shared_ptr` ensures *communal ownership*, which is well-defined. Thus, handling `shared_ptr` can sometimes feel as comfortable as worry-free object management in Java. However, there are fundamental differences. Especially because can precisely determine the timing of resource release, `shared_ptr` in C++ has an advantage over garbage collection.

20.6 Raw Pointers

Whenever I have shown you an example with raw pointers, I have tried to make sure to discourage you from using them. That was intentionally phrased a bit too strictly. You can (or even should) use such a pointer when the pointer does not *own* the object, meaning it is not responsible for its removal via `delete`:

```
int getSize(const Rect *r) { return r->size(); } // raw pointer as argument
int main() {
    Rect srect{8,12};
    cout << getSize( &srect );           // get address of stack object
    unique_ptr<Rect> urect{new Rect{10,20}};
    cout << getSize( urect.get() );      // get raw pointer from smart pointer
}
```

It goes without saying that you could just as well have used a reference as a parameter here:

```
int getSize(const Rect &r) { return r.size(); } // Reference as argument
```

In that case, you would only need to adjust the calls a bit.

What you definitely should not do is to put the result of `new` into a raw pointer. You will still see it often in traditional C++ code, but I will go so far as to mark it as an error here—in the context of this book and considering the fact that you want to learn *modern* C++, mind you.

```
// https://godbolt.org/z/Y4sMdzde8
struct StereoImage {
    Image* right_;           // ⚡ raw pointer
    Image* left_;           // ⚡ raw pointer
    StereoImage(const string& nameBase)           // construct
        : right_{new Image{nameBase+"right.jpg"}} // okay
        , left_{new Image{nameBase+"left.jpg"}}  // dangerous
    { }
    ~StereoImage() {           // remove
        delete right_; delete left_;
    }
    StereoImage(const StereoImage&) = delete;     // no copy
    StereoImage& operator=(const StereoImage&) = delete; // no assignment
};

int main() {
    Image* image = new Image{"image.jpg"};      // ⚡ a raw pointer?
```

```
StereoImage stereo{"3d"};
    delete image;
}
```

Listing 20.7 When a raw pointer owns an object, the potential errors are often hard to detect.

The raw image pointer stores the result of `new`. The corresponding `delete` is at the end of `main`. But what happens if this were another function in your program and an exception was thrown before `delete`? `delete` would never be executed, and you would have a memory leak. Always use smart pointers here:

- `auto image = make_unique<Image>("image.jpg");`—also `unique_ptr`
- `auto image = make_shared<Image>("image.jpg");`—also `shared_ptr`

Maybe you might say: “Then I’ll just catch the exception and call `delete` afterward.” Yes, that would be a start. But that won’t save you during the initialization of `links_`: an exception in the constructor of `Image` during the initialization of `links_` has very subtle effects. The `new` for `right_` has already been executed and the memory requested. An exception during the initialization of `left_` does prevent memory from being lost for `left_`, but `right_` is irretrievably lost. Because the exception leaves the constructor, `stereo` is considered not created—and thus the destructor with the `delete` statements is not called. You have no access to `right_` to clean it up. You have a memory leak.²

I recommend using smart pointers here as well:

- `unique_ptr<Image> right_;` `unique_ptr<Image> left_;` or
- `shared_ptr<Image> right_;` `shared_ptr<Image> left_;`

And then initialize them with `right_ { new Image{ nameBase+"right.jpg" } }, left_ { new Image{ nameBase+"left.jpg" } }`. The problems are solved.

Another advantage is that you don’t necessarily have to delete the copy constructor and assignment operator with `= delete`:

- With `unique_ptr`, these are impossible, and the compiler cannot generate them anyway—so you don’t have to explicitly forbid them.
- With `shared_ptr`, you get both operations, and if you use them, they do what they are supposed to do and don’t break your program.

You can read more about deleting methods in [Chapter 16](#).

When raw pointers manage an object, they should not own it. However, to refer to an object that belongs to someone else, they are indeed very useful. You saw in [Listing 20.1](#) using `*` in a container to modify the element in place. You have already encountered the `&` reference, a language element that allows you to refer to an element in the

² That is not entirely correct: the obscure language feature of the *function try block* could save you, but you would be one of the very few C++ programmers who use it.

container. But that would not have worked in the example. Because as practical and safe as a `&` reference is, it cannot do two things that a pointer can:

- **You cannot reassign a reference**

When you write `int value = 42; int &ref = value;`, `ref` will *always* refer to `value` from now on. So with `ref = 99;`, you are always assigning something new to `value`. However, if you have an `int *ptr = &value;` pointer, you can achieve the same thing with `*ptr = 66;` as with the reference, but you can also point to a completely new variable. The following would not work with references:

```
int value = 42;
int &ref = value;
ref = 99;           // value is now 99
int *ptr = &value;
*ptr = 66;         // value is now 66
int new_value = 73;
ptr = &new_value;  // ptr no longer points to value
*ptr = -1;         // value remains 66, new_value is now -1
```

- **You cannot store references in a container**

There can be no `vector<int&>` or similar. A reference is something abstract that only exists for the compiler. In the compiled program and at runtime, you no longer see any trace of it. However, every container needs a physically present piece of memory for its elements. So here you need some form of pointer—be it a smart pointer, raw pointer, or even iterator, as you will see later.

Here, raw pointers certainly have a field of application. If the objects in the container are not owned by the pointers, then `unique_ptrs` are out of the question anyway. `shared_ptrs` would be possible, but why should you manage shared ownership when it does not exist? Raw pointers are just the right thing for that.

```
// https://godbolt.org/z/W4W61h4Kv
#include <vector>
#include <numeric>    // iota
#include <iostream>
using std::vector; using std::cout;
struct Number {       // representative of a large, expensive object
    unsigned long val_;
    Number(unsigned long val) : val_{val} {}
    Number() : val_{0} {}
};

/* determines whether z is a prime number based on previous prime numbers */
bool isPrime(const Number& z, const vector<Number*> primes) {
    for(Number* p : primes) {
```

```
        if((p->val_*p->val_) > z.val_) return true;    // too large
        if(z.val_ % p->val_ == 0) return false;        // is a divisor
    }
    return true;
}
int main() {
    vector<Number> allNumbers(98); // 98 zero-initialized elements
    std::iota(begin(allNumbers), end(allNumbers), 3); // 3..100
    /* allNumbers now contains {3..100} */
    vector<Number*> primes{};      // stores determined prime numbers
    Number two{2};
    primes.push_back(&two);        // the 2 is needed
    for(Number &z : allNumbers) {  // iterate over all numbers
        if(isPrime(z, primes)) {
            primes.push_back( &z ); // store address
        }
    }
    /* Output the rest */
    for(Number* p : primes)
        cout << p->val_ << " ";
    cout << "\n";
}
```

Listing 20.8 “primes” contains pointers to another container.

I have already shown you how to quickly calculate a list of prime numbers in [Chapter 8](#), [Listing 8.1](#). In principle, it works the same way here. In `isPrime()`, I use the previously calculated prime numbers in `primes` to check if they divide `z`. If the square is greater than `z`, you have checked all potential divisors and found a new prime number.

This time, however, I decided not to store a *copy* in the result vector, `primes`. Assume `struct Number` was something that would be very expensive to copy. Therefore, `primes` takes the addresses of the `Number` elements—that is, the raw pointers.

For each determined prime number, use `&z` to get the address of the `Number` object and append it to `primes`:

```
primes.push_back( &z );
```

In terms of types, this means the following:

- `z` is a `Number`;
- `&z` is therefore a `Number*`;
- `primes` is a `vector<Number*>`; and therefore,
- The elements of `primes` are of type `Number*`.

You will find the `&z` sequence twice in the example. I would like to point out that it has different meanings depending on the context. `&z` in `push_back(&z)` means “address of `z`,” but in the `for(Number &z : ... loop`, it reads as “`z` is of type reference to `Number`.” Another interesting point is the following:

```
Number two{2};
primes.push_back(&two); // the 2 is needed
```

Because `allNumbers` initially contains only numbers starting from 3, you don't have an object in `allNumbers` for the first prime number—2—whose address you can store in `primes`. But you don't need that with pointers. You can take the addresses of objects that are located anywhere in memory. The `Number two{2};` variable is an object with an address that immediately goes into `primes`...

You see that when storing addresses, you are by no means limited to obtaining your raw pointers from just one source. You just need to make sure that all sources exist as long as the stored numbers.

So what should you absolutely avoid doing? `primes` returning as a result, like in [Listing 20.9](#). There, the source objects with `allNumbers` were destroyed upon leaving the function. The pointers you stored in `primes` for `return primes;` are no longer valid outside the function. Dereferencing them is then prohibited and at best leads to a program crash.

```
vector<Number*> primePointers(unsigned long upto) { // ⚡ Vector of * is suspicious
    vector<Number> allNumbers;
    vector<Number*> primes{};
    // ...
    for(Number &z : allNumbers)
        if(isPrime(z, primes))
            primes.push_back( &z ); // store address
    return primes; // ⚡ Pointers to function-local objects
}
```

Listing 20.9 Do not return addresses of function-local objects.

20.7 C-Arrays

With `vector` and `array`, you already have very good options for storing multiple similar elements in sequence. But sometimes you need to go back to the basics, and in this case, that means C-arrays.

In principle, a C-array is the same as a raw pointer with the additional information of a *size*. While a pointer points to a single value, a C-array points to the beginning of several consecutive values:

```
int value = 42;
int* pointer = &value; // points to a single int
int carray[10] = { 1,2,3,4,5,6,7,8,9,10 }; // C-array of 10 int values
```

Now `carray` has type `int[10]`, and you can iterate over all elements using the range-based `for` loop:

```
for(int val : carray)
    cout << val;
```

You can also read the fifth element or write to the eighth element. It should be noted that, as with `vector` and `array`, the index of the elements starts at zero:

```
int x = carray[4]; // get the 5th element
carray[7] = 12;    // write to the 8th element
```

How a C-array works with raw pointers is a bit trickier. For example, you can implicitly convert any C-array to a simple pointer of the same type. This gives you an `int*`:

```
int* ptr = carray;
```

This `ptr` now points to `carray[0]`. So with `*ptr = 99;`, you set `carray[0] = 99;`.

20.7.1 Calculating with Pointers

When a raw pointer points to a C-array, you can perform arithmetic with this pointer. For example, use the `++` `--` operations as well as `+` and `-` like you would with an integer.

```
// https://godbolt.org/z/a5b6qMo9h
#include <iostream>
int main() {
    int carray[10] = { 1,1 }; // initialized to { 1,1,0,0,0,0,0,0,0,0 }
    int* end = carray+10;    // pointer past the last element
    for(int* p = carray+2; p != end; ++p) {
        *p = *(p-1) + *(p-2); // adds the previous two numbers
    }
    for(int const * p=carray; p != end; ++p)
        std::cout << *p << " ";
    std::cout << "\n";
}
```

Listing 20.10 With raw pointers pointing to a C-array, you can perform arithmetic.

This program outputs `1 1 2 3 5 8 13 21 34 55`. How does that work? First, the program initializes `carray` with two ones, followed by as many zeros as needed. *Warning:* If you do not specify an initialization, the C-array contains random values—for example:

```
int carray[10]; // ✗ C-array remains uninitialized
```

With `carray+10`, you initialize `end` so that it is a pointer *behind* the last element of the C-array. It is a common convention in C and C++ to remember the end of a range in this way, rather than pointing to the last element itself. The side effect is that the difference between the two `carray` and `end` pointers is exactly the number of elements in between: `((end-carray)==10)` is true.

Then initialize the `int*` pointer `p` for the loop with `carray+2`, thus pointing to the first element after the two ones in `carray`. This should be the first position you want to write to. For each iteration of the loop, advance `p` by one position in the C-array. This is easily done with `++p`. However, you need to check the `for` loop after each increment to see if the loop has ended: `p != end` checks if `p` is not yet pointing to the element after the C-array, which would signify the end of the loop. Thus, during the loop, `p` points to `carray+2`, `carray+3`, `carray+4`, ... up to `carray+9`, to execute the loop body. The final increment advances `p` to `carray+10`, which represents the loop's termination condition.

In the loop body, you write to `*p`, which means what `carray+2` to `carray+9` refer to sequentially. What do you write? Well, `*(p-1) + *(p-2)` for the sum of the two previous elements—for the first iteration, effectively:

```
*p = *(p-1) + *(p-2); // in the first loop iteration, p = carray+2, so:
*(carray+2) = *(carray+2-1) + *(carray+2-2); // which is:
*(carray+2) = *(carray+1) + *(carray+0);      // which is:
carray[2] = carray[1] + carray[0];             // so:
carray[2] = 1 + 1;
```

Here you have learned about the different notations. `*(carray+x)` is equivalent to `carray[x]`.

In the second loop iteration, when `p = carray+3`, it ultimately results in

```
carray[3] = carray[2] + carray[1]; // that is:
carray[3] = 2 + 1;
```

and so on and so forth, until the entire `carray` is overwritten to `{1, 1, 2, 3, 5, 8, 13, 21, 34, 55}`.

20.7.2 Decay of C-Arrays

You have seen that you can iterate over a C-array with the range-based `for` loop. You can do this because, unlike a raw pointer `int*`, the C-array `int[10]` carries its size with it. Unfortunately, this information is very volatile, and as soon as you pass the C-array to a function, it is lost. The C-array then *decays* to a simple raw pointer, which cannot remember the array size. Specifically, `int[10]` (and any other size) becomes an `int*`.


```
// https://godbolt.org/z/1qx16Gr61
#include <iostream>
void fibonacci(int data[], int* end) {
    for(int* p = data+2; p != end; ++p) {
        *p = *(p-1) + *(p-2);
    }
}

std::ostream& print(std::ostream &os, int data[], int* end) {
    for(int const * p=data; p != end; ++p)
        std::cout << *p << " ";
    return os;
}

int main() {
    int carray[10] = { 1,1 }; // initialized to { 1,1,0,0,0,0,0,0,0,0 }
    fibonacci(carray, carray+10);
    print(std::cout, carray, carray+10) << "\n";
}
```

Listing 20.11 C-arrays decay to raw pointers as parameters.

For this reason, you cannot declare the parameter of a function as `int[10]`, but only as `int*`. However, you can give a hint to the users of the function. For `int*`, you can use the alternative notation `int[]`. This indicates that the parameter originates from a C-array of `int` elements. However, you can only communicate the size of the array in the comment. Or—and this is the usual way in these cases—you add an end parameter to the function, which (as in the previous example) points to the end of the C-array.

Apart from the fact that I now use `int data[]` as a function parameter instead of `carray[10]` directly, not much has changed. The additional parameter `end` is needed to indicate the end of the loop.

As an alternative to `end`, you can of course also pass the number of elements—here, 10—when calling:

```
std::ostream& print(std::ostream &os, int data[], size_t count) {
    for(int const * p=data; p != (data+count); ++p)
        std::cout << *p << " ";
}
```

In the C environment, this variant is used more frequently, while the version with the end pointer is more practical when dealing with the C++ standard library. As you will see, these insights about pointers can be directly transferred to the concept of *iterators*.

It is best to use `int const * p` where you do not intend to change the values pointed to by the loop variable. If you want to change the values, you should use `int* p`.

20.7.3 Dynamic C-Arrays

When you declare a C-array as an automatic stack variable, the size of the array must be a constant—or `constexpr`, to be precise:

```
void calc(size_t sz) {
    int carray[10]; // okay
    int darray[sz]; // ✗ Error: sz is not a constant
}
```

If you want to write the latter, you need a compiler that supports *dynamic automatic C-arrays on the stack* (standard since C++14). Until then, you need to create such a *dynamic C-array* with `new[]`:

```
void calc(size_t sz) {
    int[] darray = new int[sz];
    // ...
    delete[] darray;
}
```

You can see that just as `new` requires `delete`, you need `delete[]` for `new[]`. *Warning:* You must not accidentally remove a C-array created with `new[]` by using `delete` (without brackets).

Once you have created such a dynamic C-array, you can interact with it just like the previously discussed automatic C-array. The only difference in handling is that you need to take care of disposal using `delete[]`.

Is `delete[]` too tedious for you? I agree with you. Just like with self-managed raw pointers, resources can be lost here if you are not very careful. You should consider whether you can use `vector` instead. And if that doesn't work, then wrap the C-array in a `unique_ptr` immediately after creation. It has the ability to call `delete[]` for you.

```
// https://godbolt.org/z/h3q839x8r
#include <memory>      // unique_ptr
#include <iostream>    // cout

std::unique_ptr<int[]> createData(size_t sz) {
    return std::unique_ptr<int[]>(new int[sz]);
}

void fibonacci(int data[], int* end) {
    for(int* p = data+2; p != end; ++p) {
        *p = *(p-1) + *(p-2);
    }
}
```

```
std::ostream& print(std::ostream &os, int data[], int* end) {
    for(int const* p = data; p != end; ++p)
        std::cout << *p << " ";
    return os;
}

int main() {
    std::unique_ptr<int[]> data { createData(10) };
    data[0] = 1; // set values in the array through the unique_ptr
    data[1] = 1;
    fibonacci(data.get(), data.get()+10); // get the C-array pointer with get()
    print(std::cout, data.get(), data.get()+10) << "\n";
}
```

Listing 20.12 “unique_ptr” works with the dynamic C-array.

As you can see, you can use everything you already know about `unique_ptr` here. You can use it as a return value and don't have to worry about disposal. With `data.get()`, you can access the underlying pointer to the C-array, and with `data[n]`, you can even access individual elements directly through `unique_ptr`.

20.7.4 String Literals

A specific C-array form is especially important because it is particularly common: the `const char[]` C-array.

If you write a text as a literal directly into the source code, then the compiler interprets it as a C-array of type `char` with a size corresponding to the length of the text. However, one character at the end is invisible: every string literal ends with the characters `'\0'` (`char` with the numeric value zero) to mark its end.

So you write the following, quite correctly:

```
const char hello[3] = "hi";
```

The `hi` C-array is three characters long. It contains the `char` elements `{ 'h', 'i', '\0' }`. Therefore, you can alternatively write it in a more cumbersome way:

```
const char hello[3] = { 'h', 'i', '\0' };
```

This is completely equivalent, but is rarely done for obvious reasons.

And you can simplify something else. When you specify a literal in this way during the definition of a variable, the compiler can determine the size of the C-array. You don't have to count it yourself. Just write the following:

```
const char hello[] = "hi";
```

This also defines the variable `hello` as `const char[3]`.

Whether you use the array brackets `[]` or the pointer asterisk `*` to declare your text constants rarely matters in practice. Choose what suits you better. Perhaps a bit unusual is the fact that the array brackets are placed after the *variable* (even though they actually belong to the type of the variable), while the pointer asterisk is placed after the type—for me, the more intuitive notation:

```
// https://godbolt.org/z/s6sss481E
const char vimes[13] = "Samuel Vimes"; // const char[13]
const char colon[] = "Fred Colon";    // const char[11]
const char* nobby = "Nobby Nobbs";    // const char[12]
```

When you use such a text literal together with `string`, you won't even notice that it is a `const char[]`. There is a constructor that converts `const char*` to `string`:

```
// https://godbolt.org/z/oKsz7d55T
#include <string>
#include <iostream>           // cout
using std::string; using std::cout;

string greet(string name) {
    return name + "!";        // string operator+(string, const char*)
}

int main() {
    string name{ "Havalock Vetinari" }; // explicit: string(const char*)
    cout << "Angua";              // ostream& operator<<(ostream&, const char*)
    cout <<                       // ostream& operator<<(ostream, string)
        greet("Carrot Ironfoundersson"); // implicitly: string(const char*)
}
```

Listing 20.13 With “`string`”, you often don't even notice that text literals are “`const char[]`”.

While `cout << "Angua"` directly outputs the C-array, with `cout << greet(...)` it is the returned `string`. The parameter of `greet` is implicitly converted from a C-array to a `string`. For the return value of `greet()`, the existing `string` is concatenated with `const char[]` using `operator+`—resulting in a new `string`.

20.8 Iterators

Container

Pointers can be considered a special case of iterators, which in turn are part of containers. Iterators are explained in detail in [Chapter 24, Section 24.2](#). Nevertheless, I want to show you how pointers interact with containers because containers like `vector`, `array`, and `string` have already been mentioned.

If you already know everything about raw pointers and C-arrays, then it's also time to consider the principles of *iterators*. Ultimately, this is quite simple: What the raw pointer is for a C-array, the iterator is for the standard containers—with the difference that you can do more with iterators and standard containers than with pointers and C-arrays. This is because iterators are implemented as classes that you can equip with additional intelligence using the usual C++ means. And raw pointers also are *compatible* with iterators in the sense that you can use raw pointers where iterators are required. One could also say that raw pointers are just a specific form of iterators. So, iterators are a concept that raw pointers fit into.

If you want to compare raw pointers with iterators, you first need to know the counterpart to get the beginning of the `int carr[10]` C-array. With raw pointers, it's simple: `carr` is implicitly converted to an `int*` and is the beginning. For standard containers, you get the beginning with the `begin()` method or the `begin(container)` free function. And where there is a beginning, in this case, there is also an end: you get it with the `end()` method or `end(container)`. This works for all standard containers, from `list` to `vector` and `set` to `array` and `unordered_map`, and so on.

```
// https://godbolt.org/z/4nqz4ed4W
#include <vector>
#include <iostream> // cout

using std::vector;
int main() {
    vector data{ 5,4,3,2,1 };
    vector<int>::const_iterator end = data.end(); // or end(data)
    for(vector<int>::const_iterator it = data.begin(); it!=end; ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";
}
```

Listing 20.14 Retrieve iterators with “begin” and “end”.

You can also increment an iterator with `++`. The loop end check is also done with a comparison, here with `it!=end`. You can also access the actual value with the `*` dereference operator, here with `*it`. What pointers can do, an iterator must also be able to do. Therefore, you could also determine the number of elements with `end(data)-begin(data)` or `distance(begin(data),end(data))`.

No sooner said than I need to add a caveat: only *most* iterators are capable of forming differences. There is an additional function called `difference()` from `<iterator>`, which is even better suited for this purpose.

But where `int const *` used to be now stands `vector<int>::const_iterator`. Because the exact type of an iterator depends on both the type of the container and the type of the elements, you need to “reach into” the container to get the exact type of an iterator. The equivalent of `int*` is `vector<int>::iterator`. You can also use `auto` for this.

20.9 Pointers as Iterators

Once you get used to iterators, you won't need to readjust for handling C-arrays. For all algorithms from the standard library that take iterators as parameters, you can also use raw pointers. Why? Because raw pointers are just a specific form of iterator. They are thus compatible.

The beginning of a C-array is a raw pointer that can be used like an iterator. This happens as follows with `int[6]` and thus behaves almost as if it were a `vector<int>`.

```
// https://godbolt.org/z/zsqKz76cj
#include <iostream>    // cout
#include <iterator>    // ostream_iterator
#include <algorithm>    // copy, ranges::copy

int main () {
    int data[6] = { 1, 2, 3, 7, 9, 10 };
    std::ostream_iterator<int> out_it (std::cout, ", ");
    std::copy(data, data+6, out_it);           // Pointers as iterators
    std::cout << "\n";                        // Output: 1, 2, 3, 7, 9, 10
    std::ranges::copy(data, out_it);           // C-array pointer as range
    std::cout << "\n";                        // Output: 1, 2, 3, 7, 9, 10
}
```

Listing 20.15 You can use raw pointers like iterators.

Here it is `copy` that now takes raw pointers as start and end positions instead of iterators. By the way: the aforementioned `distance` from `<iterator>` works on pointers just like all other such functions. And as you can see when calling `ranges::copy`, pointers also work with C++20 ranges.

20.10 Pointers in Containers

You can also put all forms of pointers into standard containers. However, the same rules apply as for pointers outside of containers; that is, ownership semantics are determined by the type of pointer you choose, not by the type of container:

- `vector<Planet*> planets` only holds the pointers themselves and does not feel responsible for the heap objects without your intervention. If `planets` is automatically removed as a stack object, `delete` is not automatically called for the elements. You must either do this yourself beforehand if `planets` is to be the owner of the objects, or keep the addresses in another variable that is then the owner.
- `vector<unique_ptr<Planet>>` is always automatically the owner of the elements, just like `vector<Planet>` would be.
- `vector<shared_ptr<Planet>>` is partially the owner of the elements, similar to a regular `shared_ptr`.

For those who have to manage without C++11 and its smart pointers, the rule is this: there can be no `vector<auto_ptr>`. You must not and cannot put an `auto_ptr` into a standard container. This is because the ownership semantics of `auto_ptr` are misleading and noncanonical—and this is the main reason that this type is *deprecated*.

20.11 The Exception: When Cleanup Is Not Necessary

There are indeed cases where you do not want to clean up objects you have created. If you know that the program is about to end anyway, cleaning up the data would unnecessarily consume time, and the only resource being occupied is memory, so you can use heap objects to save time.

Every stack object and every global object normally would be removed upon exiting `main`. But maybe you have created an enormous neural network or a super large search tree. If, for example, it consists of a million small nodes, then cleaning it up can take a few seconds. At the end of the program, the operating system releases all memory en bloc anyway. So you can save yourself the trouble of releasing it just before the program ends.

And this is the case where you are allowed to use `new` without `delete`. If the program is going to terminate soon anyway, you can save yourself the trouble of cleaning up memory-only resources.

```
// https://godbolt.org/z/xE1TWx1ve
#include <map>
#include <memory> // unique_ptr
#include <string>
#include <iostream>
```

```

#include <chrono> //time measurement
using std::map; using std::cout; using std::endl; using namespace std::chrono;
struct Node {
    std::unique_ptr<int> d_;
    Node() : Node{0} { }
    explicit Node(int d) : d_{ new int } { *d_ = d; } // also some memory
    friend bool operator<(const Node& a, const Node& b) {return *a.d_<*b.d_;}
    friend bool operator==(const Node& a, const Node& b) {return *a.d_==*b.d_;}
};
long long millisSince(steady_clock::time_point start) { //Helper for measurement
    return duration_cast<milliseconds>(steady_clock::now()-start).count();
}

int main() {
    std::unique_ptr<map<int,Node>> huge{ new map<int,Node>{} };
    cout << "Building..." << endl;
    steady_clock::time_point start = steady_clock::now();
    for(int idx=0; idx < 100*1000*1000; ++idx) { // massive amount in the map
        (*huge)[idx] = Node{idx};
    }
    cout << "Done: " << millisSince(start) << " ms" << endl; //timing here
    start = steady_clock::now();
    huge.reset(); // cleanup here
    cout << "End: " << millisSince(start) << " ms" << endl; //timing here
}

```

Listing 20.16 Is it worth saving the release of memory?

To measure how long something takes after exiting `main()`, you would need timing outside the program. Here, I simulate the cleanup of the map by calling `huge.reset()`, telling `unique_ptr` to clean up its charge. You would have done the same without smart pointers with the following:

```

map<int,Node>> *huge = new map<int,Node>{};
// ...
delete huge;

```

But here you want to avoid raw pointers. In both cases, the rule is this: if you omit `delete huge`; or write `huge.release()`; instead of `reset()`, you save the time needed to clean up the many `Node` objects. You can find out how much time you save at "End:".

At "Done:", `millisSince` determines the time in milliseconds required to build the data structure. I chose the number `100*1000*1000` (i.e., 100 million) so that my computer can just handle it:

- Building it takes my computer about 52,000 milliseconds with nine gigabytes of memory.³
- Tearing it down takes 6,700 milliseconds.

It's not really much—but on other computers and under different conditions, it might take more time. Whether you need this time-saving feature when exiting is up to you. However, do not let the program design suffer because of it.

In the `millisSince()` helper function, I use `steady_clock` from `<chrono>` for precise time measurement.

3 2.9 GHz i7-3520M, 16 GB memory, g++ 4.8.

Chapter 21

Macros

Chapter Telegram

- **Preprocessor**

A program that runs before the actual C++ compilation phase; nowadays, it is usually not implemented as a separate program but as a phase of the compiler.

- **Macro**

An identifier in the source code that is textually replaced by something else during the preprocessor phase; typically distinguished from a define by having additional arguments.

- **Define**

A macro without arguments that is simply marked as *is now present* or replaced by simple text.

- **Include guard**

A define that prevents the same header file from being accidentally included multiple times by the same .cpp file.

In C++, there are many mechanisms that the compiler can use to assist in programming. Above all, the type system supports you—when used correctly—in writing more robust programs. There is a mechanism that bypasses the type system because it is executed *before* all other tasks of the compiler: the *preprocessor*—from *pre-*, meaning *before*.

You can use the preprocessor primarily for the following three different tasks. However, in modern C++, only `#include` still has full justification. This is because, especially since C++11, there are better alternatives for everyday programming within the core C++ language:

- **Include other files with `#include`**

With `#include <header.hpp>`, you include the interface of another module so that you can use its functions.

- **Conditional compilation**

With `#if` and `#ifdef`, you can completely exclude certain areas so that the C++ compiler no longer sees them. You sometimes need this to get your code running on multiple platforms. But the blessing is also the curse: code that the compiler no longer sees you will probably forget about and no longer maintain. If possible, you

should write your code to be portable without the preprocessor by using the portable features of the standard library. Ideally, you should only use `#ifdef` as include guards (see [Chapter 12](#)). If you are writing a portable library, you cannot avoid it.

■ Macros

You can define textual replacements with `#define`. These can even have parameters, similar to functions. Even though macros are a powerful tool, you should use (inline or template) functions, constants with `const`, `constexpr`, and `enum class`, as well as custom types or `using/typedef` wherever possible instead. Type safety is greater, error diagnosis is simpler, and the code is generally easier to understand. Some exceptional cases confirm this rule.

21.1 The Preprocessor

Preprocessor directives always begin with the hash symbol `#` at the start of a line. The entire line is then an instruction for the preprocessor to evaluate. You can add a comment with `//` or `/* */`. A line ending with a backslash `\` (without a space after it) continues on the next line as if the line break were not there.

```
// https://godbolt.org/z/j6WWxPheP
// Filename: my-macros.hpp
#ifndef MY_MACROS_HPP // Include Guard
#define MY_MACROS_HPP
#include <iostream> // cout, cerr
#include <vector>

#ifdef OUTPUT_TO_STANDARD
# define OUT std::cout
#else
# define OUT std::cerr
#endif

#define MESSAGE(text) { (OUT) << text << "\n"; }
using container_type = std::vector<int>;
static constexpr unsigned SIZE = 10;
#endif
```

Listing 21.1 Lines containing preprocessor directives start with a `#`.

You can now include this header. For example, the next listing shows how your main file with `main` looks.

```
// https://godbolt.org/z/feGrM4f6j
// Filename: makros.cpp
#define OUTPUT_TO_STANDARD // Switch between cerr and cout
#include "my-macros.hpp"
#include "my-macros.hpp" // Oops, accidentally included twice.
int main() {
    MESSAGE("Program start");
    container_type data(SIZE);
    MESSAGE("The container has " << data.size() << " elements.");
    MESSAGE("Program end");
    OUT << "That was a close call.\n";
}
```

Listing 21.2 Your main file with “main” includes the header file via “#include”.

All preprocessor actions take place purely on a textual level before the compiler actually comes into play. Under Linux, you invoke the compiler for `makros.cpp` like this (adjust accordingly for other systems):

```
g++ -o macros.x macros.cpp
```

You read the source code from the `macros.cpp` file and output the fully translated program to `macros.x`.

The preprocessor reads the `makros.cpp` file line by line. The first line is `#define OUTPUT_TO_STANDARD`. This introduces the `OUTPUT_TO_STANDARD` identifier to the preprocessor. You simply introduce it as a name whose existence you can later check with `#ifdef` or `#ifndef`.

Defines on the Command Line

Instead of writing `#define OUTPUT_TO_STANDARD` in the source code, you can also inform the compiler during translation that a specific define should be set. How to do this depends on the compiler. For `g++`, this is done with the `-D` switch on the command line. So you could have started the translation like this:

```
g++ -DOUTPUT_TO_STANDARD -o macros.x macros.cpp
```

It follows `#include "my-macros.hpp"`. The preprocessor reads the file completely and immediately replaces the `#include` directive. Then it continues line by line with the result—starting with the content just read from `my-macros.hpp`.

The first thing there is:

```
#ifndef MY_MACROS_HPP
#define MY_MACROS_HPP
```

With `#ifndef`, you instruct the preprocessor to include the following block if the `MEINE_MAKROS_HPP` preprocessor variable does not exist and ignore it if it does. `ifndef` stands for *if not defined*. The world of the preprocessor is quite different from that of C++. It doesn't really care about C++ identifiers. If you had used names like `vector` or `std` here (bad idea!), they would not yet be known to the preprocessor. Only things that were previously made known to the preprocessor, such as with `#define`, are relevant here.

This did not happen with `MY_MACROS_HPP`: `#ifndef MY_MACROS_HPP` is true. This causes the preprocessor to process all text up to the corresponding `#endif` or `#else`.

This is followed by another `#include` of the standard library and then a check to see if the preprocessor recognizes `OUTPUT_TO_STANDARD`. Between `#ifdef`, `#else`, and `#endif`, there are two alternatives, one of which the preprocessor chooses, and it completely ignores the other—as if the lines were empty. Because I specifically defined `OUTPUT_TO_STANDARD` with `#define`, the `#ifdef` is true, and the first alternative `#define OUT std::cout` remains.

Note here that I actually set `#define OUTPUT_TO_STANDARD` in a different file than the `#ifdef OUTPUT_TO_STANDARD` check. However, it is precisely the purpose of the preprocessor to merge the various files. So it doesn't matter that the two occurrences come from different files. The preprocessor merges everything into a continuous stream.

The `#define OUT std::cout` introduces the new name `OUT` to the preprocessor. This time, it is not just a pure introduction of a name, but with `std::cout`, the replacement text `std::cout` is desired. From now on, the preprocessor will replace every occurrence of the token `OUT` with `std::cout`. The preprocessor has similar word separation rules as C++, so `SHOUT` naturally does not become `SHstd::cout`—but `std::OUT` does become `std::std::cout`. The preprocessor does not recognize the scope resolution operator `::`.

Then follow two simple C++ definitions that are included in the output stream. `container_type` is now available as a type alias, and `SIZE` is a numeric constant.

From the included header, one relevant line remains:

```
#define MESSAGE(text) { (OUT) << text << "\n"; }
```

This also introduces a preprocessor identifier. From now on, every occurrence of `MESSAGE` will be replaced by `{ (OUT) << text << "\n"; }`. However, the preprocessor expects `MESSAGE` to be used with a parameter, which it calls `text`. The text provided as a parameter is inserted completely and unchanged into `{ (OUT) << text << "\n"; }` exactly as you specified it in the call.

Back in `macros.cpp`, this means that the line

```
MESSAGE("the container has " << data.size() << " elements.");
```

will become the following:

```
{ (std::cout) << "the container has " << data.size() << " elements." << "\n";
}
```

OUT is replaced by the previous definition here and text by the MESSAGE argument.

When the preprocessor has finished its work, it results in a single data stream. This is then used for the actual C++ compilation. Apart from the includes of the standard library, the result after the preprocessor and before the C++ compiler is as shown in [Listing 21.3](#).

```
// https://godbolt.org/z/W45G5TjTz
// ...here content of <vector>...
// ...here content of <iostream>...
using container_type = std::vector<int>;
static constexpr unsigned SIZE = 10;
int main() {
    { (std::cout) << "Program start" << "\n"; }
    container_type data(SIZE);
    { (std::cout) << "The container has " << data.size() << " elements." << "\n"; }
    { (std::cout) << "End of program" << "\n"; }
    { std::cout << "That was a close call.\n"; }
}
```

Listing 21.3 The result of the preprocessor run.

After this phase, nothing remains of the preprocessor directives. The preprocessor variables are also completely replaced by their equivalents. There are some interesting points to note:

- The textual substitution of the preprocessor occurs exactly. It leaves every semicolon, parenthesis, and bracket intact and does not introduce any new ones. You can see this especially in MESSAGE: the block brackets {...} and the semicolon are present in the result. text could also contain multiple output elements separated by <<, which the C++ compiler can later interpret appropriately.
- container_type and SIZE are not preprocessor identifiers.

Use “constexpr if” for Compile-Time Code Dependencies

In C++17, an addition made it so that the compiler can discard an if branch from the program. This way, you need to use less preprocessor syntax to switch between code variants and can stay entirely within C++.

```
// https://godbolt.org/z/8j9j9KoWx
#include <iostream> // cout, cerr
constexpr int OUTPUT_VARIANT = 0; // Switch between cerr and cout
```

```

void message(const char* text) {
    if constexpr(OUTPUT_VARIANT == 1)
        std::cerr << text << "\n";
    else
        std::cout << text << "\n";
}

int main() {
    message("Message");
}

```

Listing 21.4 With “constexpr if”, you can switch between variants without a preprocessor.

Here, the compiler decides at translation time which of the two `if` branches to include in the code.

21.2 Beware of Missing Parenthesis

It is particularly important that the preprocessor does not introduce or remove parentheses but only performs a textual replacement, especially when macro parameters are used in the replacement expression.

```

#define SQUARE(x) x*x
int result = SQUARE(3+4)

```

is evaluated to 19, as you can see here. However, `SQUARE(3+4)` should probably be `7*7`, which is 49. This is very dangerous:

```
int result = 3+4*3+4
```

Normally, you should *always* enclose each use of a parameter in the replacement text with brackets. And often, the entire replacement text also needs to be enclosed in brackets.

```

#define SQUARE(x) ((x)*(x))
int result = SQUARE(3+4)

```

becomes what you want, because the following calculation results in 49:

```
int result = ((3+4)*(3+4))
```

Exceptional cases like text from [Listing 21.1](#) only confirm the rule. There, omitting the parentheses in the macro definition allows you to chain multiple stream operations `<<` in the parameter when calling the macro.

21.3 Feature Macros

Throughout this book, I have occasionally noted that a particular feature is only available from C++17, C++20, or C++23 onward. And from time to time, I mentioned that some compilers do not yet support a feature. From C++20 onward, there is an official way to check if the compiler or the standard library supports a particular feature. You can then respond to such conditions with `#if` or `#ifdef`. These feature macros are then defined and have a numerical value like 201707L:

```
#if __cpp_generic_lambdas >= 201707L
    // Here you can use generic lambdas with template parameters.
#else
    // Here you have to do without them.
#endif

#ifdef __cpp_lib_as_const
    // Here there is std::as_const(coll).
#else
    // Here not.
#endif
```

The list for the compiler currently includes 67 macros, and the library's list includes 214 macros.

21.4 Information about the Source Code

Sometimes it is interesting for analysis purposes to know in which file and line the execution is currently taking place. For this, there are the `__FILE__`, `__LINE__`, and `__func__` macros.

With C++20, this is now part of the standard library. With the `<source_location>` header, you can get the necessary information via `source_location::current()`:

```
#include <source_location>
#include <iostream>
int main() {
    const auto loc = std::source_location::current();
    std::cout << "In " << loc.file_name() << " in line " << loc.line()
        << " in function " << loc.function_name() << "\n";
}
```

This currently outputs the following for me:

In 11pModularisierung.md in line 687 in function int main()

Normally, you will see a `.cpp` file here. But because I extract the C++ code from my Markdown document and insert corresponding hint lines as part of this book project, I get a suitable output in my actual source document. Here you can see how I use `#line` in the previous listing to refer to my actual document and not the temporary `.cpp` file (excerpt):

```
#line 682 "11pModularisierung.md"
    #include <source_location>
#line 685 "11pModularisierung.md"
    #include <iostream>
#line 686 "11pModularisierung.md"
    int main() {
#line 687 "11pModularisierung.md"
        const auto loc = std::source_location::current();
#line 688 "11pModularisierung.md"
        std::cout << "In " << loc.file_name() << " in line " << loc.line()
            << " in function " << loc.function_name() << "\n";
```

So when you generate code, you can use the `#line` directive to inform the compiler of the actual source file name for error messages and `source_location`.

21.5 Warning about Multiple Executions

And in another place, you might not get what you wanted at first glance. Compare the effects of the following two implementations.

```
// https://godbolt.org/z/jzKfabPh3
#include <cmath> // sin, cos
constexpr double max2(double a, double b) { return a > b ? a : b; }
#define MAX2(a,b) ((a) > (b) ? (a) : (b))
int main() {
    double f = max2(sin(3.141592/2), cos(3.141592/2));
    double e = MAX2(sin(3.141592/2), cos(3.141592/2));
}
```

Listing 21.5 The purely textual substitution leads to multiple executions in macros.

For the `max2` function, the arguments are evaluated *before* entering the function. The expensive `sin()` and `cos()` calculations therefore take place beforehand. In `max2`, only simple numerical values need to be compared.

The `MAX2` macro, however, expands to the following:

```
double e = ((sin(3.141592/2)) > (cos(3.141592/2)) \
    ? (sin(3.141592/2)) : (cos(3.141592/2))));
```

Now, `sin()` and `cos()` are complicated functions that the compiler cannot precompute. It is very unlikely that the compiler is intelligent enough to avoid executing `sin()` or `cos()` twice. A function is definitely the better choice here.

More disastrous than just a loss of time is the multiple execution you can get when expanding the arguments:

```
// https://godbolt.org/z/YzYz3x1zx
#include <iostream>
#define MAX2(a,b) ((a) > (b) ? (a) : (b))
int main() {
    int x = 0;
    int y = 0;
    int z = MAX2(++x, ++y); // ⚡ expands arguments multiple times
    std::cout << "x:"<< x << " y:"<< y << " z:"<< z << '\n';
}
```

What is the output here? Attention, this is a trick question! For me, the output is this:

```
x:1 y:2 z:2
```

The macro expanded the expression to `((++x) > (++y) ? (++x) : (++y))`. And thus `++y` was executed once in the comparison and once in the result. Do you want that? Absolutely not, because it is *undefined behavior* to execute two side effect operators on the same variable in the same expression. So if you see a different output or the program crashes, that's "allowed."

21.6 Type Variability of Macros

However, the `MAX2` macro implementation has one advantage over the `max2` function. You can also use the macro for types other than `double`. Because the macro is used at the call site, the types of the variables are determined at the time of use. Therefore, `MAX2` also works on `int` and even on `string`:

```
// https://godbolt.org/z/5zrfvovaW
#include <string>
#include <cmath> // sin, cos
#define MAX2(a,b) ((a) > (b) ? (a) : (b))
int main() {
    double e = MAX2(sin(3.141592/2), cos(3.141592/2));
    int i = MAX2(10+12+45, 100/5+20);
    std::string s = MAX2(std::string("Ernie"), std::string("Bert"));
}
```

However, type variability is better achieved in C++ with overloading. You can define `max2` for each type you use:

```
// https://godbolt.org/z/6xzTx83zE
#include <string>
#include <cmath> // sin, cos
constexpr double max2(double a, double b) { return a > b ? a : b; }
constexpr int max2(int a, int b) { return a > b ? a : b; }
std::string max2(const std::string &a, const std::string &b)
    { return a > b ? a : b; }

int main() {
    double e = max2(sin(3.141592/2), cos(3.141592/2));
    double i = max2(10+12+45, 100/5+20);
    std::string s = max2("Ernie", "Bert");
}
```

As you can see, in the case of `max2("Ernie", "Bert")`, it has even more advantages. First, the function does not reevaluate the arguments, and second, you save the explicit type conversion from `const char[]` "Ernie" to a `std::string` with `std::string("Ernie")`. The function `max2()` takes `string` as arguments, and the compiler recognizes that it can convert `const char[]` to `string`.

Perhaps you want to implement `max2` for `string` but with more intelligence than for `int` and `double`—possibly without distinguishing between uppercase and lowercase. As a macro, there can only be a single `MAX2`, and with overloaded functions, you can make each implementation look different if necessary.

However, if the body of all `max2` functions is identical, then the necessary duplicate code is cumbersome and does not comply with the guidelines for good code. Therefore, I would like to provide you with a recipe that shows you how to avoid using macros in this case—as that would be even less in line with the guidelines for good code (according to the rules of this book).

If it comes to the point where you need to implement a function identically for all possible (or at least several) types, write the function as a simple *function template*. This is a preview, but don't worry, it's quite simple:

- First, write one of the implementations with a specific type like `string`.
- Take the type that changes in all implementations and replace it with the word `TYPE`.
- Write `template <typename TYPE>` in the line before the function.

Done. It looks like the following listing for `max2`.

```
// https://godbolt.org/z/acMjvxcM5
#include <string>
#include <cmath> // sin, cos
```

```
template<typename TYPE>
constexpr TYPE max2(const TYPE &a, const TYPE &b)
{ return a > b ? a : b; }

int main() {
    double e = max2(sin(3.141592/2), cos(3.141592/2));
    double i = max2(10+12+45, 100/5+20);
    std::string s = max2(std::string("Ernie"), std::string("Bert"));
}
```

Listing 21.6 A simple function template is much better than a macro.

Congratulations! You have written your first function template. This implementation is as flexible as a macro, as fast as overloaded functions, and most importantly, type-safe.

And starting from C++20, even *abbreviated function templates* are possible:

```
constexpr auto max2(const auto &a, const auto &b) { return a > b ? a : b; }
```

If you want, you can even add overloads for individual types. And that's exactly what you should do if you want to get rid of the annoying explicit conversions to `std::string` for the arguments "Ernie" and "Bert". If you omit them here, the compiler will try to find an overload for `max2(const char[6], const char[5])`. It won't succeed because they are two different types—but in the template, I used only one `TYPE`, and it must match exactly in both places. So the template does not match the call. Therefore, simply add another overload:

```
const char* max2(const char* a, const char* b) {
    return std::string(a) > std::string(b) ? a : b;
}
```

This also immediately solves the problem that comparing two `const char*` is by no means a comparison of the texts, but only *of the addresses* of the pointers. Converting to `string` specifically for the comparison with `>` solves the problem—albeit not elegantly. Unfortunately, `>` for `string`, unlike for `int`, is not `constexpr`, and the compiler thus does not allow the use of `>` on `string` in a function marked as `constexpr`. But at least you now have a template for almost all cases—and an extra overload for a useful special case.

C++20: “consteval” and “constexpr” Bring Even More Macro Replacements

With `constexpr`, you can mark functions from C++20 onward that the compiler can guarantee to compute at compile time. Normally, it will then use the result at the point of use. With `constexpr`, you can declare variables from C++20 onward that the compiler guarantees to compute at compile time.

By the way, you can save yourself the conversion of the `const char[]` literal because the compiler automatically interprets string literals with the `"s` suffix as `std::string`. So you can write the following:

```
std::string s = max2("Ernie"s, "Bert"s);
```

However, you must have included `operator"s` beforehand—for example, with using namespace `std::literals`.

21.7 Summary

For clarity, [Table 21.1](#) lists the most important preprocessor directives once again. I have added a few more with a brief explanation.

Directive	Explanation
<code>#include</code>	Include another file
<code>#define</code>	Define identifier or macro
<code>#undef</code>	Remove identifier definition
<code>#ifdef</code>	Check if a preprocessor identifier is defined
<code>#ifndef</code>	Check if a preprocessor identifier is not defined
<code>#elifdef</code>	Alternative check if an identifier is defined (C++23)
<code>#elifndef</code>	Alternative check if an identifier is not defined (C++23)
<code>#if</code>	Check preprocessor expression
<code>#else</code>	Alternative source code to one of the preprocessor <code>ifs</code>
<code>#elif</code>	Alternative source code and another check
<code>#endif</code>	End of the alternative source code
<code>#line</code>	Set current line number and filename for compiler error messages
<code>#error</code>	Abort with an error message
<code>#pragma</code>	Special instruction to the compiler; very compiler-dependent

Table 21.1 Overview of possible preprocessor directives.

Chapter 22

Interface to C

Chapter Telegram

- **C library with C interface**

Extension often provided by third parties.

- **extern "C"**

Mark a function so that it can be used by C++ programs.

- **void* or void pointer**

Non-type-safe raw pointer type, to and from which all raw pointer types can be converted; often part of C interfaces.

- **Custom delete function or custom deleter**

Feature of `shared_ptr`, which can be useful for releasing C resources.

If you are writing a program that uses third-party libraries, then there is a good chance that these libraries come with a C interface. Fortunately, C++ is written in such a way that you can definitely include libraries for C as well. So your choice is not limited.

What is limited is your choice of language features. If you want to use a C library, you can only use C language elements. Here is a by no means exhaustive list of restrictions that you have to accept:

- You have no classes. Structures can only have certain elements; for example, constructors are omitted, and everything must be public. For simplicity, you should only have data elements.
- There are no references. You pass a value or raw pointer.
- Many things from the C++ standard library are taboo. You cannot use standard containers, strings, or streams, but the C++ language designers have thought about interaction possibilities.
- C functions do not throw exceptions. You usually deal with error codes as return values or parameters.
- There are no overloaded functions in C; each name has a fixed set of parameter types (this has partially changed with C11). `operator` functions do not exist either.
- You will rarely encounter `const` with parameters. When `const` is present, it is more for documentation purposes. You cannot be sure that the C function does not modify your parameter passed by pointer.

After this theoretical list, we dive into practice. As an example, I have chosen the *zlib*¹ C library, which I want to integrate.

The *zlib* library contains functions for *compressing* data. This means that redundant information is removed, leaving a smaller package of data that you can, for example, transfer over the internet or store on a hard drive. A group of functions in the library is used for reading and writing *.gz files—a widely used data format. If you have a program on your computer for packing or unpacking files (e.g., 7zip or WinZip), you can most likely also unpack *.gz files. One difference of *.gz compared to other packing formats is that each packed file may contain only exactly one original file. It is common to simply append .gz to the original file name when the file has been compressed.

22.1 Working with Libraries

A C library that you include consists of the following components:

- **Library file**

You *link* your program with the library file to use the third-party functions. These are one or more *.a, *.dll (Windows), or *.so files (Linux). In rare cases, you also receive the source code and link the *.o/*.obj object files to your program.

- **Header files**

You are usually provided with several *.h files. You need to include these in your source code with `#include`.

- **Documentation**

Sometimes you get a separate document; sometimes you only have the header.

How you make the library and header known during translation is very compiler-dependent. Refer to the documentation on how to adjust the *library* and *include path* and how to link a specific library.

With g++, this example is translated as follows:

```
g++ gzpack.cpp -o gzpack.x -lz -Lzlib -Izlib/include
```

You should already be familiar with the `g++ gzpack.cpp -o gzpack.x` part. The remaining parameters are as follows:

- **-lz**

-l means that a library should be linked. Here it is *z* for *libz*.

- **-Lzlib**

-L specifies search paths for libraries; here *libz.a* is located in the *zlib* directory.

¹ “A Massively Spiffy Yet Delicately Unobtrusive Compression Library,” <http://zlib.net>, [2014-05-21]

- **-Izlib/include**

-I specifies search paths for header files; here `zlib.h` is located in the `zlib/include` directory.

22.2 C Header

The demonstration program creates a compressed output file from any input file. Only three functions from `zlib` are needed for this. The documentation contains a lot of information about those functions, which I recommend you read. I focus here on the information in the header file or the function signatures from the documentation (some things are simplified). For your own experiments and to replicate the examples, you should of course use the original library:

```
typedef struct gzFile_s *gzFile;
extern "C" gzFile gzopen (const char *path, const char *mode);
extern "C" int  gzwrite (gzFile file, voidpc buf, unsigned len);
extern "C" int  gzclose (gzFile file);
```

These and many other types and functions are available to you when you include the following in your source code:

```
#include <zlib.h>
```

First, the typedef introduces the new `gzFile` type. This is a pointer to `struct gzFile_s`. Its internal details are intentionally not mentioned because they are irrelevant. Consider `gzFile` simply as a handle that allows you to access the *.gz file.

The three functions are free functions, and all parameters and returns are either built-in types or raw pointers—very common in C.

Each function is introduced with `extern "C"` so that as a C++ author, when you include that header, you can distinguish C++ functions from C functions. This is necessary because C++ uses different mechanisms when searching for the appropriate function; this is important for overloading and the like. So, coming from C++, every function signature you want to use must be marked with `extern "C"`. If this is missing in a header, you can alternatively include it in your source code in a slightly more cumbersome way:

```
extern "C" {
    #include <zlib.h>
}
```

So you wrap the entire `#include` in a large `extern "C"`.

After you have the header, you continue with the rest of the program. In addition to `<zlib.h>`, you also need some of the usual C++ headers from the standard library. For error handling with `zlib`, the `<cerrno>` and `<cstring>` C headers are necessary. The former allows querying the global error code variable, `errno`, which `zlib` uses; this is stated in the documentation. The latter provides `strerror` to convert an error code into a readable text message. Both headers are part of the C standard library. Therefore, each header exists in two variants: In C++, you use `<cerrno>` when `<errno.h>` is specified in the C documentation. Similarly, the C include `<strings.h>` becomes `<cstring>` for C++ programs. The “prefix with `c`, remove the `.h`” rule applies to most headers of the C standard library—but only to those, not to third-party libraries. The C++ variants are part of the C++ standard library and guarantee, for example, that each function also gets its extern “C”. In addition, all C functions imported this way are provided in the `std::` namespace. Therefore, you can also call the function via `std::strerror()`.

```
// https://godbolt.org/z/9Trh8h8Mb
#include <string>
#include <vector>
#include <span>      // C++20
#include <fstream>    // ifstream
#include <stdexcept> // runtime_error
#include <iostream>   // cerr
// C-Header:
#include <zlib.h>      // gzXYZ; sudo aptitude install libz-dev
#include <cerrno>      // errno
#include <cstring>     // strerror
namespace {
using std::string; using std::span; using std::byte;
```

Listing 22.1 The “`gzpack.cpp`” program uses a C library. Here you see the section with the includes.

In the second-to-last line, I start with `namespace {`—an *anonymous namespace*. This hides all variables, types, and functions in this `*.cpp` file from potential conflicts with others. Instead of free functions, you could also write `static`, but for data types, this is only possible through an anonymous namespace, which is often a good idea.

With the two `using` declarations, I save typing effort for the rest of the source code. Another advantage of the anonymous namespace is that the two `using` declarations also apply only within the namespace and do not affect the outside.

Next, I define the class that bundles the main functionality of the C library. In this example, all three relevant C functions are already included. Based on the names, you can probably already guess the meanings and some of the operating rules:

- `Gzopen()` opens a file. If it fails, the function returns a special value according to C conventions.
- `Gzclose()` closes a correctly opened file. The call is not necessary if `gz-open` failed.
- `Gzwrite()` writes data to the output. A return value greater than zero indicates success.

```
// https://godbolt.org/z/Yh7j1WEGc
class GzWriteStream {                                // RAI wrapper
public:
    gzFile gz_ ;                                     // C struct from zlib.h
    explicit GzWriteStream(const string& filename)
        : gz_{gzopen(filename.c_str(), "wb9")} // 'w': write, 'b':binary, '9':level
    {
        if(gz_==NULL) throw std::runtime_error(std::strerror(errno));
    }
    ~GzWriteStream() {
        gzclose(gz_);
    }
    GzWriteStream& operator<<(const span<char> &data) {
        write(data);
        return *this;
    }
private:
    void write(span<char> data) {
        auto bytes = std::as_bytes(data);           // C++20
        auto res = gzwrite(gz_, bytes.data(), size(bytes));
        if(res==0) throw std::runtime_error("Error writing");
    }
    GzWriteStream(const GzWriteStream&) = delete;    // no copy
    GzWriteStream& operator=(const GzWriteStream&) = delete; // no assignment
};
```

Listing 22.2 This part of “gzpack.cpp” contains all the C functions used.

As a data field, `GzWriteStream` contains the `gz_` handle as a connection to `zlib` and thus to the opened file. In the constructor's init list, `gzopen()` opens the file and assigns the result to the handle. The documentation states that a return value of `NULL` indicates an error. Then the global `errno` variable shows the exact error. You can decode this code with `strerror`. More importantly, you should also throw an exception and exit the code this way, instead of jumping out with a `return` or running to the end.

If `gzopen()` fails, then `gzclose()` is not necessary. Therefore, this is an optimal use case for using the destructor. If an exception leaves the constructor instead of running normally, the `GzWriteStream` object is considered not created. Thus, no destructor is called—and no `gzclose()` either.

The first parameter that `gzopen()` receives is the file name as a C string. From the file-name parameter of `string`, you can obtain the file name as a C string using the `c_str()` method. The second parameter `"wb9"` indicates that the file is opened for writing ('w' for *write*). The 'b' stands for binary format. The alternative 't' for text format would potentially convert individual characters. The final '9' stands for maximum data compression.

22.3 C Resources

Packing the `gzopen()` and `gzclose()` function pair together in the constructor and destructor of a class is particularly suitable here, as these two always need to be called in pairs. There is no (successful) `gzopen()` without `gzclose()`, and no `gzclose()` without `gzopen()`.

You deal with such pairings in many C interfaces. Whether it's initializing the entire library or acquiring and releasing a paintbrush for the window frame—such function pairs are common, and such a wrapper class is then appropriate.

In `GzWriteStream`, I have included even more functionality. Sometimes, however, you only need the pairing of two functions. Then it might be tedious to write a separate wrapper class for each pair. If that is too much work for you, you can fall back on your old acquaintance `shared_ptr`: you can use it in the variant where the constructor takes not one but two arguments.

The second argument here is a *function object*, which is called within the destructor of `shared_ptr`. `shared_ptr` calls this a *custom deleter*.

I will make a brief digression here, because as a function object you see a *lambda*—a function without a name—and within it `gzclose()` is called:

```
std::shared_ptr<gzFile_s> gz(  
    gzopen(fNameOut.c_str(), "wb9"),  
    [=](gzFile_s* f) { gzclose(f); } );  
if(gz.get() == NULL) throw std::runtime_error("Error");
```

You will learn more about this in [Chapter 23, Section 23.3](#).

You can almost always use this pattern with the `shared_ptr` constructor with two arguments for simple pairings of C functions.

22.4 “void” Pointers

The third function, `gzwrite()`, takes two parameters: a `void*` to the data and the length of the data. The return value is the amount of data written; a value of zero indicates an error.

But what exactly is `void*`? Did you notice that our own `write()` function, using the `as_bytes` function, created a `span<byte>` whose `data()` method returns a `const byte*`, which I then passed to the `void*` of `gzwrite()`? Normally, C++ always complains when you take a variable of one type and try to use it as something else. Imagine if `gzwrite()` took an `int`: then the `const byte*` from `write` wouldn't fit, and I would have had to define the parameter differently—or use a trick. And what if the data to be written are `int*` data or `double*`? Do you then need a `gzwrite()` function for each pointer type? Yes and no: `gzwrite()` doesn't really care whether the pointer points to `byte`, `unsigned char`, `int`, or `double` data, so long as you also provide the correct length of the data. And this is exactly where `void*` comes into play: all raw pointer types can be converted to a `void*` without the compiler complaining. What the respective function then does with these absolute raw data is entirely its own business. Here, it takes the raw data, whose length it knows, compresses it, and then writes it to the output file.

In a C function, it is more common to pass raw data of unspecified types as parameters. Often, you also pass the length of the raw data, and sometimes you pass a specific marker that describes the data in more detail. Less frequently, you don't need to provide any additional information. In that case, the address is only stored for later use, or the exact meaning is derived from the context.

When you work with `void*`, you unfortunately lose all type safety. The compiler accepts all raw pointers when converting to and from `void*` and doesn't even warn you. Why would it? It assumes you know what you're doing when working with `void*`. I hope this will continue to be the case for you: use `void*` only when dealing with C programming interfaces. In the C++ world, you have type-safe means to handle variant data, such as function overloading or templates.

22.5 Reading Data

The following listing presents two simple helper functions that handle reading and writing.

```
// https://godbolt.org/z/84TcTbWjj
std::vector<char> readFile(const string& fName) {
    std::ifstream file{ fName, std::ifstream::binary };
    if(!file) throw std::runtime_error("Error opening input");
    file.seekg(0, file.end);           // jump to the end of the file
    const auto length = file.tellg(); // current position is file size
    if(length > 1024*1024*1024)
        throw std::runtime_error("No more than 1 GB please");
    file.seekg(0, file.beg);           // back to the beginning
    std::vector<char> data(length);    // allocate space
    file.read(data.data(), length);    // read in one go
```

```
    return data;                                // not copied (keyword: RVO)
}
void pack(const string& fNameIn, const string& fNameOut) {
    auto data = readFile(fNameIn);              // read input
    GzWriteStream gz{fNameOut};                 // initialize output
    gz << data;
}
```

Listing 22.3 This part of “gzpack.cpp” handles reading and writing the files.

It is somewhat tricky to read the data in one go into a `vector<char>`. First, `tellg()` determines the position of the end of the file, which was previously jumped to with `seekg()`. Another `seekg()` brings the read position back to the beginning so that `file.read()` reads from the start of the file. There could have been other ways to determine the file size, but they are not necessarily less complicated than this.

The `vector<char>::data()` function returns a `char*`, into which `ifstream::read()` reads the data directly into the vector. However, the vector must have enough space allocated beforehand; that’s why you create it with the `length` constructor argument in the appropriate size.

Besides `vector`, only `array` has such a `data()` method. No other standard container guarantees that its data is contiguous, allowing it to be read in one go with `read()` and similar functions.

A word about `return data;`, where the return type is `vector<char>`—a value, not a reference or pointer. Potentially, returning such a value type could require a large copy operation. However, the compiler and the standard library work together here to “avoid the copy” (*copy elision*). This is guaranteed in this case because all `return` statements within the function return the same local variable; here it’s trivial because there’s only a single `return`. In this case, the compiler never needs to create a copy for the return, even if it is returned by value. Before the standard mandated copy elision in this situation, this scenario was commonly referred to as *return value optimization* (RVO).

The `pack()` function starts with two trivial lines. The first line, `readFile()`, reads the input, and the second line, `GzWriteStream gz`, prepares the output. This is done using the overloaded operator `<<`. For simplicity, I implemented it as a method. A free friend function would have been possible as well, but it is not necessarily needed here as the stream class is under my control—unlike with `ostream`, where you can only work with a free function overload of `operator<<`.

22.6 The Main Program

The conclusion is then formed by `main()` from [Listing 22.4](#). You will not encounter any surprises. You need to close the anonymous namespace opened in the first listing, then `main` begins.

A comprehensive `try-catch` handles any exceptions that you might have thrown elsewhere in the program. `exc.what()` outputs an appropriate error message.

The `fNamen` vector stores all command-line arguments, which the `for` loop then iterates over. In it, `pack()` calls the compression function.

```
// https://godbolt.org/z/5bWx6PzdP
} // namespace
int main(int argc, const char* argv[]) {
    try {
        const std::vector<string> fNamen {argv+1, argv+argc};
        for(auto fName : fNamen) {
            std::cout << "packing " << fName << "... ";
            pack(fName, fName+".gz");
            std::cout << fName << ".gz" << "\n";
        }
    } catch(std::runtime_error &exc) {
        std::cerr << "Error: " << exc.what() << "\n";
    }
}
```

Listing 22.4 With “`main`” in “`gzpack.cpp`”, the example is complete.

22.7 Summary

Let’s conclude with a brief recap:

- Virtually every third-party extension consists of a library and headers.
- Include the header(s) with `#include`.
- Look at the parameters and return types of the functions and choose the appropriate C++ ones.
- Texts in `const char*` or `char*` can usually be provided using `string`.
- Large dynamic data sets with a raw pointer interact well with `vector`, `unique_ptr`, or `shared_ptr`.
- Specifically, look out for function pairs for resource management and consider managing them with a wrapper class or a custom deleter function via `shared_ptr` using C++ language features.

Chapter 23

Templates

Chapter Telegram

- **Template**
A class template or function template.
- **Function template**
A template that takes a type as a template parameter to become a function.
- **Template function**
A complete function that is instantiated from a function template.
- **Class template**
Synonym for a parameterized type.
- **Template parameter**
A type that is fixed at compile time for a template; instead of a type, a template parameter can also be a constant number.
- **Parameterized type**
A class that takes another type (or a constant number) as a template parameter.
- **Instantiate template**
The moment of using a template with specific types (or constants); the compiler generates the program code at this moment.
- **Concept**
The `concept` language element allows templates to be constrained. This can make templates more consistent and generally leads to more understandable compiler error messages.
- **Function parameter, callback, or callback function**
A function that is passed as a parameter to a “worker function” and then called within the worker function.
- **C function pointer**
Pass a function parameter by address to a previously defined function.
- **function<>**
The C++ type of a function parameter.
- **Function object or functor**
A class that defines `operator()`, or an instance of it.
- **Anonymous function or lambda expression**
A locally defined function without a name.

- **Capture clause**
List of outer variables that you can access from within a lambda expression.
- **Formal data type**
The type parameter in a template declaration.
- **User-defined literal**
When you override `operator""`, you define a suffix for custom character or numeric literals.
- **Fold expression**
Within a template with a variable number of arguments, the use of an operator like `+` or `|` together with `...`, which the compiler automatically concatenates.

I won't hide it: templates are difficult. But I want to make it easy for you, because if you completely avoid templates, you will miss out on most of the good things in C++.

The good news: You don't need to fully understand templates to use them. I would even argue that there are few people who have completely understood templates. However, you can still use them without too much mental gymnastics and thus get the most benefit from them. In fact, *you have* already used templates—namely, `string`. And I very much hope that you have come to know and love `string` by now. The second faithful companion through most pages of this book, with whose help you have already used templates, is `vector`—in its `vector<int>`, `vector<double>`, or even `vector<Car>` form.

So you have already dealt with the first aspect of templates: you use them as *parameterized types*. And that's exactly what a *template* is: a class that takes another type as a parameter is a *class template*. `vector` is one of them, and so is `basic_string`. Give `basic_string` a `char` as a parameter, and you get this:

```
using string = basic_string<char>;
```

Voilà! This is simplified, because `basic_string` can do a bit more. I will explain how to create templates—the class templates—once you have learned about function templates.

23.1 Function Templates

The second aspect of templates is function templates. Here, it is not a new class that takes a type as a parameter, but a function. You will learn this part just as quickly as using `string` and `vector`, because what you will be doing here is also not new: you are overloading functions. I will begin the explanation with this topic.

23.1.1 Overloading

Strictly speaking, a function parameter consists of two parts: the *value* and its *type*. At runtime, you can call a function with different *values*: sometimes with 5, sometimes with 42, sometimes with -10. However, the *type* always remains the same.

A function where the type itself is variable is called a *function template*: Here, you must specify the type of a parameter in addition to its value when calling.

But first, let's consider a normal function like this:

```
void print(int value) {  
    cout << value;  
}
```

You call it at runtime with the following:

```
print(5);  
print(42);  
print(-10.25);
```

Here, the *value* of the parameter changes. Even if you pass a double value like in `print(-10.25)`, `void print(int)` is still called, and the parameter -10.25 is converted to an `int` value -10 for the call. So, for the last line, -10 would be output instead of -10.25.

This is not a problem for you, because you can write a function overload for the double type:

```
void print(double value) {  
    cout << value;  
}
```

With this additional definition of `print(double)`, you also get the output -10.25. What do you get for the following line?

```
print("Flamingo");
```

You get an error message stating that there is no suitable overload for `const char[]`. Said and done:

```
void print(const char* value) {  
    cout << value;  
}
```

And now `print("Flamingo");` works as well. But I'm starting to notice that the function body is always the same. It always says `{ cout << value; }`. If the program gets more types that you want to output to `cout` with `print()`, it becomes tedious. You also get a lot of code duplication, and you should avoid that.

23.1.2 A Type as Parameter

If you write it in such a way that the *type* is also flexible, it looks like this:

```
template<typename TYPE>
void print(TYPE value) {
    cout << value;
}
```

One could say the function now has two parameters: `TYPE` for the type of the argument and `value` for its value.

When calling it specifically, you now need to specify *both* things. As such, you write the type argument after the function name in angle brackets:

```
print<int>(5);
print<double>(-10.25);
print<const char*>("Flamingo");
```

But the compiler already knows that 5 is an `int`, -10.25 is a `double`, and "Flamingo" is a `const char*`. And when the compiler can make the choice for you, you can also omit the angle brackets and the explicit type when calling this function:

```
print(5);
print(-10.25);
print("Flamingo");
```

These lines are equivalent to the previous example. However, it is important to note that you now actually have three functions: one for `int` as a parameter, one for `double`, and one for `const char*`.

Note that whether you write `template<typename TYPE>` or `template<class TYPE>` is irrelevant. Both notations are equivalent, and it is a matter of taste or agreement to use one or the other. The standard library prefers `class` here, while I personally prefer `typename`.

23.1.3 Function Body of a Function Template

The function body of `print` takes advantage of the fact that the output operator `operator<<` is defined for `int`, `double`, and `const char*`. This is, of course, not true for all types.

```
// https://godbolt.org/z/aMPj3K7EE
#include <iostream>
struct Number {
    int value_;
};
template<typename TYPE>
void print(TYPE value) {
    std::cout << value << "\n";
}
```

```

}
int main() {
    print(5);
    print(-10.25);
    print("Flamingo");
    Number seven { 7 };
    print(seven);    // ✗ cout << seven does not exist
}

```

Listing 23.1 The call to a function template works so long as the function body is valid with the template parameters.

The compiler also generates a new function here—essentially:

```

void print(Number value) {
    std::cout << value;
}

```

But for `Number`, there is no `operator<<`. Therefore, the compilation of the `print(seven);` statement fails.

There is an important term you should remember: you *instantiate* a function template at the moment you actually call it with a specific type. This is important because it is only at that moment that the compiler checks whether the template's source code can be translated.

For example, you can make the listing compilable after the fact by adding:

```

std::ostream& operator<<(std::ostream& os, Number z) {
    return os << z.value_;
}

```

Think of it roughly as the compiler looking at the content of the function only at the moment you use it concretely. This “looking at” consists of the concrete type parameter being applied at all points and the function being truly generated—as if you had written it into the source code. This may not be a completely accurate perspective, but it is useful and sufficient for getting started.

If your terminology is to be precise, then use the following terms:

- A *function template* is the blueprint.
- You *instantiate* the function template by specifying the template parameters.
- Then it becomes a *function*—more precisely, a *template function*.

The last distinction is only important in the context of overloads. When the compiler tries to determine the correct function for a call, it first goes through the *real functions*

before considering the *template functions*. An instantiated function template ranks below normal functions without type parameters in the overload resolution hierarchy.¹

```
// https://godbolt.org/z/816Ys17aT
#include <iostream>
using std::cout;
template<typename TYPE>
    void func(TYP a) { cout << a << " TYP\n"; }
void func(int a) { cout << a << " int\n"; }
int main() {
    func<int>(8); // Output: 8 TYP
    func(8);      // Output: 8 int
}
```

Listing 23.2 With and without explicit type specification during the call.

This program first outputs 8 TYP because the *template function* was explicitly requested with angle brackets.

However, both definitions would fit the second call. If there is both a matching *function* and a matching *template function*, the *function* is preferred during overload resolution.

C++20: Abbreviated Function Template

For the simple way to define a function for any parameter types, C++20 offers that you write an *abbreviated function template*. You can then omit `template<...>` and instead use `auto` as the argument type:

```
#include <iostream>
void show(auto a) { std::cout << a; } // no template<...>
int main() {
    show(8);
    show(6.66);
}
```

Here, the compiler recognizes from `auto` as the argument type that you actually mean and interprets it as the following code:

```
template<typename T> void show(T a) { std::cout << a; }
```

So, using `auto` as an argument type also introduces a function template.

¹ Such a distinction is not necessary for classes. Therefore, the terms *template class* and *class template* are not used distinctly here.

23.1.4 Values as Template Parameters

In most cases, it is types that are template parameters of functions. However, there are indeed cases where the parameter is a value. You can use numbers like 10, characters like 'a', and since C++20 also floating-point numbers like 12.89 as well as strings like "abc" and instances of *structure types* (literal and public) like `array` or `pair` as template parameters.

You already know this with classes. `array` takes a constant number as its second parameter. `array<int, 10>` creates space for exactly 10 values of type `int`. What happens if you want to write a function that sometimes creates an array of size 10 and sometimes an array of size 100? You already know that this doesn't work because `n` is a variable and not a constant—or to be precise, not a `constexpr`:

```
array<int, n> createArray(size_t n);
```

But it almost works this way. Turn `createArray` into a function template and `n` into a template parameter instead of a simple function parameter, as in the next listing.

```
// https://godbolt.org/z/c9xzM7EhP
#include <array>
#include <iostream> // cout
using std::array; using std::cout;
template<size_t SIZE>
array<int, SIZE> createArray() {
    array<int, SIZE> result{};
    return result;
}
int main() {
    auto data = createArray<5>();
    data[3] = 33;
    for(auto e : data) cout << e << " ";
    cout << "\n";
}
```

Listing 23.3 A template parameter can also be a constant number.

The output will unsurprisingly be 0 0 0 33 0. The mechanism is exactly the same as for types as parameters: the moment you use the function, you instantiate it. The compiler replaces all occurrences of the template parameter with what you specified at the call, and you get a completely new function. Essentially, the instantiated function for `createArray<5>()` looks like this:

```
array<int, 5> createArray() {
    array<int, 5> result{};
    return result;
}
```

And because `array<int,5>` as opposed to `size_t n = 5; array<int,n is valid, the example works.`

Also, `auto` works for values as template parameters. Then the value can take any valid type. However, this feature is rarely used.

```
template <auto value> void f() { }  
f<10>();           // value gets type int  
f<'a'>();          // value gets type char
```

Listing 23.4 With `auto` as a template parameter, you can specify any simple value.

Together with the variable number of template arguments `...`, you can specify any number of different types and values as template parameters. Also, `auto` together with `...` is rarely needed.

```
template<auto ... vs> struct MixedList {};  
using Three = MixedList<'a', 100, 'b'>;  
Three three{};
```

Listing 23.5 With “`auto`” and `...`, you can specify any values as template parameters.

23.1.5 Many Functions

Instantiating a function template actually means that the compiler generates the entire function: function header and function body. If you call—sorry, *instantiate*—a function template with 20 different template parameters within a `*.cpp` file, then you actually get the function body 20 times, even if the function body always looks the same. However, the machine code that is generated often looks very different—for example, for `print(5)` and `print("Flamingo")`. The output of `int` is certainly implemented very differently from the output of a string. For this reason, the compiler needs all these function variants (or template functions, as previously explained).

But don't worry: if the compiler notices that 18 of these variants produce the same machine code and only two variants are really different, then it will compress the duplicates of the functions before assembling the overall program. This is not a must, but any reasonably modern compiler is capable of doing this.

However, this “unrolling” of the function body at the moment of instantiation of the function has a huge advantage: the function body is tailored to the place of the call. The compiler can use this and actually save the *call* of the function. It can directly insert the function body of the function templates where you actually called a function.

This *inlining* has considerable speed advantages. An executed function call is always a huge slowdown during the runtime of any program. Not only do parameters have to be prepared, but the jump into the function and out of the function must also be executed. For current CPUs, a function call is like a construction site on the highway:

internal memory areas can only be slowly refilled, planned overtaking maneuvers have to be aborted, and so on.

All in all, it is often very worthwhile if the compiler saves a function call. And function templates are the ultimate sign to tell the compiler to “inline, please.” Here, the compiler puts in a lot of effort.

23.1.6 Parameters with Extras

Some functions use the “pure” formal type of the template, while others qualify it:

```
template<typename TYPE>
TYPE add(TYPE a, TYPE b) {           // use the pure type
    return a + b;
}

template<typename TYPE>
void print(const TYPE& value) { // type enriched with const and &
    cout << value;
}
```

It depends on what you want to do with the parameters in the function. Because the `add()` function probably only makes sense for numbers, it will very likely only be applied to built-in types. Thus, passing parameters by value is very sensible: simply use `TYPE`.

The `print()` function, on the other hand, looks like it could be used for all sorts of types—and you certainly want to avoid a copy if possible. By qualifying `TYPE` with `const &`, you ensure that the function generated by the compiler also declares the parameter as `const &`—for a by-reference parameter passing.

Establishing a general rule for when to use what is difficult. After all, you might overload `operator+` for a custom data type. And if you then call your new `add()` template—say, for `Matrix`—then a by-value parameter might be very costly.

Therefore, I want to give you the following guidelines:

- If you are quite sure that your function template will only be called with very specific types, write the function—regarding the enrichments—as you would without templates.
- If you don't have control over the type arguments, prefer to perform a calculation with a newly generated return value by value, as with `add`.
- Choose a reference (not `const`) if you want to modify the parameters. This is rather rare with templates, as you don't know exactly what type you will get.
- You will often have `const` references, because if you only need to read, that is the most flexible option.

However, I don't want to hide from you that dealing with qualifiers is sometimes not that simple. Imagine that the `TYP` template parameter is completely replaced by the type that is present at the moment of instantiation. So if you already have a constant reference (i.e., `const&`), then `TYP` is also a constant reference when instantiated.

```
// https://godbolt.org/z/fev879zaK
#include <iostream>
const int& a_or_b(int choice) {
    static const int a = 42;
    static const int b = 73;
    if(choice==1)
        return a; // return const& to inner variable a
    else
        return b; // return const& to inner variable b
}
template<typename TYPE>
TYPE add(TYPE a, TYPE b) {
    return a + b;
}
int main() {
    auto res = add(
        a_or_b(0), // const int&
        a_or_b(1) ); // const int&
    std::cout << res << "\n"; // Output: 115
}
```

Listing 23.6 For “`TYPE`”, the compiler determines “`const&`”.

Because the return value of `a_or_b` is always a `const int&`, `TYPE` for the compiler in the call to `add` in `main` is exactly this `const int&`. And because `TYPE` is also the return value of `add`, it is this `const int&`. A reference as a return value is very dangerous. Where does the reference point to? Well, in this example, fortunately, it's not a problem because `const` references are always somewhat less problematic than mutable references. In this case, the `+` has created a new value, and as we only access it as `const&`, the compiler is smart enough to retain the immutable value for the output on `cout`.

Actually, it's quite simple: the compiler replaces exactly `TYPE` with what you instantiated the template with. All embellishments that the type already has are preserved and passed through. In some cases, this can lead to unexpected results. Often, it's compiler error messages that you need to interpret, especially when using the standard library. Therefore, I wanted to be clear on this point about how the compiler proceeds, so you can correctly interpret the error messages.

You might now wonder what happens if the type is already `const&` when instantiated but—as in `print`—you also enrich it with `const&`. Well, the rules are quite simple:

- When two `const`s meet, the result is `const`.
- When two `&`s collide, the result is `&`.

So if it happens that you pass the return of `a_or_b` into `print`, the compiler instantiates `void print(const int&)`—not `void print(const const int & &)`, which wouldn't make sense.

There is another case I need to mention. When you pass a temp-value as a parameter, it is a `&&`. A temp-value is usually treated like a normal reference `&` in terms of template parameters. Here, for example, the compiler instantiates `TYPE` as `Image&`, even though this temp-value is actually an `Image&&`:

```
print( Image{"my-image.png"} );
```

Only when you write your own template where the type parameter itself is qualified with `&&` do you need to consider special cases—so very likely not for the time being. If you believe you need to write such a template to manage your project, refer to the literature on universal references and perfect forwarding as well as reference collapsing.

Until then, remember: do not write templates with `&&` as type parameters, because you might not get what you expect. If an actual argument that is a reference `&` matches a `&&` template parameter, it becomes a normal reference `&`. Only if both the actual argument is an rvalue reference `&&` and the template parameter is `&&` will the parameter in the instantiated function be an rvalue reference `&&`.

23.1.7 Method Templates are Just Function Templates

You can write a method template just as you can write a function template. Is that enough? Almost, right?

There is really nothing special about methods compared to functions, except that the first parameter with the invisible `this*` is already set, meaning its value and type. I think you just need an example.

```
// https://godbolt.org/z/bfofn76Ke
#include <iostream>
class Printer {
    std::ostream& trg_;
public:
    explicit Printer(std::ostream& target)
        : trg_(target)
    {}
    template<typename TYPE>
    Printer& print(const TYP& arg) {
        trg_ << arg;
        return *this;
    }
};
```

```
    }  
};  
int main() {  
    Printer normal(std::cout);  
    normal.print(7).print(" ").print(3.1415).print("\n");  
    Printer error(std::cerr);  
    error.print(8).print(" ").print(2.7183).print("\n");  
}
```

Listing 23.7 A method can also be a template.

I have deliberately chosen a comparable example, again with `print()`. This time, the only difference is that you need to create an object—an instance of `Print`. You decide with the constructor's parameter where the output goes. If you pass `std::cout`, then `print()` goes to the standard output; if you pass `std::cerr`, then `print()` goes to the error stream.

In the `Printer::print()` method, there is no difference from a normal function, except that you can access the `trg_` data field.

The compiler generates a separate method for each argument type with which you call `print()`. In this example, these are `print(int)` for 7 and 8, `print(double)` for 3.1415 and 2.7183, and `print(const char*)` for `"\n"`. The functionality is the same as with functions.

23.2 Function Templates in the Standard Library

In the standard library, you will find function templates everywhere. You have already seen some examples, especially from the `<algorithm>` and `<numeric>` headers.

The functions from these headers are particularly interesting because they work with the standard containers. If you pass an entire container as a parameter, starting from C++20, it becomes a *range* (see [Chapter 24, Section 24.1.7](#)). Alternatively—and before C++20—you pass a pair of iterators, as in the next listing.

```
// https://godbolt.org/z/oTn1K4sGG  
#include <vector>  
#include <iostream> // cout, ostream  
#include <algorithm> // sort, copy  
#include <iterator> // ostream_iterator  
int main() {  
    std::ostream_iterator<int> oit(std::cout, " ");  
    std::vector data { 100, 50, 1, 75, 25, 0 }; // vector<int>  
    std::sort(std::begin(data), std::end(data)); // Iterator pair  
    std::copy(std::begin(data), std::end(data), oit); // Iterator pair  
    std::cout << '\n'; // Output: 0 1 25 50 75 100  
}
```

```

std::ranges::reverse(data);           // Range
std::ranges::copy(data, oit);        // Range
std::cout << '\n';                   // Output: 100 75 50 25 1 0
}

```

Listing 23.8 You will also find various function templates in the standard library.

After `sort()`, I use `ostream_iterator` together with `copy()` to output the contents of `data` to `cout`. `ostream_iterator` is an adapter that converts each element assignment into a `<<`. `copy()` thus actually leads to stream output.

Here I used `vector<int>` and obtained the iterators from it with `begin()` and `end()`, which `sort()` and `copy()` require as parameters. These are the template arguments. And because they are template arguments, the whole thing works (almost always) with *all* standard containers and with all iterators. Instead of `vector<int>`, you could just as well have used `list<string>` or `array<double>`. You would not have had to change the rest of the program. It also works with a `vector<Image>` if you have overloaded `operator<` and `operator<<` for `Image`.

For `reverse()` and the corresponding `copy()`, I used the range variants. Here, the entire container is passed, not the iterators as a pair.

23.2.1 Ranges instead of Containers as Template Parameters

If you want to write a function that operates on a container, it is advisable to do it the same way as the standard library: do not pass the container as a parameter, but as a *range*, which has been available in the standard since C++20. The effect is that your function does not care whether the range belongs to a vector or a map.

```

// https://godbolt.org/z/h3baE65od
#include <iostream>
#include <vector>
#include <set>
#include <bitset>
#include <ranges>
namespace rng = std::ranges;
std::ostream& printBinary(std::ostream& os, rng::input_range auto&& range) {
    for(auto&& elem : range) {
        std::bitset<4> x(elem); // Copy number into bitset
        os << x << " ";
    }
    return os;
}

```

```
int main()
{
    std::vector vdata { 2, 0, 15, 12 };
    printBinary(std::cout, vdata) << "\n";
    // Output: 0010 0000 1111 1100
    std::set sdata { 2, 0, 12, 15 };
    printBinary(std::cout, sdata) << "\n";
    // Output: 0000 0010 1100 1111
    int adata[] = { 0,1,2,13,14,15 };
    printBinary(std::cout, adata) << "\n";
    // Output: 0000 0001 0010 1101 1110 1111
}
```

Listing 23.9 Pass your own algorithm ranges as arguments, not the container.

With `auto` as the second parameter, you have defined a template. Before C++17, you had to put `template<typename T>` before the function and then `T&&` for the parameter—as described in the previous section. The `&&` ensures that you can pass both values directly from variables and temp-values from expressions. Before `auto` is the `ranges::input_range` *concept*. It works without it, but you tell the compiler and the user of the function a lot when you write it down. In particular, the compiler's error messages will be better. You can find more about ranges as parameters in [Chapter 25, Section 25.6](#), under the subheading “Ranges as Parameters (and More).”

The `printBinary()` function converts all elements of the range into a `bitset<4>`, which is then printed. The conversion to `bitset` naturally only works if the container from which the iterators originate contains integer types, which is the case here with `int`. If you try `printBinary()` with a container that contains `string` or `double`, the compiler will report an error exactly where the conversion is attempted with `std::bitset<4>x(elem);`.

As for the type of container, using ranges is flexible. Try it with a vector like `vdata`, and you will get the content output as binary numbers. With a set like `sdata`, the elements are also sorted—a property of the `set` class. And because raw pointers are just a certain form of iterators, you can even call the `printBinary()` function with a C-array like `adata`. Ranges work for standard containers as well as for C-arrays.

Note that I have omitted the template arguments for vector data and set data here and let the compiler deduce them, which has been possible since C++17.

If you are programming for a compiler that does not yet support ranges, you need to “unpack” the range—that is, get the two iterators and call the function with the iterators.

```

// https://godbolt.org/z/13eMxKTfj
#include <iostream>
#include <vector>
#include <set>
#include <bitset>
template<typename IT>
std::ostream& printBinary(std::ostream& os, IT begin, IT end)
{
    for(IT it=begin; it != end; ++it) {
        std::bitset<4> x(*it); // Copy number into bitset
        os << x << " ";
    }
    return os;
}
int main()
{
    std::vector<int> vdata { 2, 0, 15, 12 };
    printBinary(std::cout, vdata.cbegin(), vdata.cend()) << "\n";
    // Output: 0010 0000 1111 1100
    std::set<int> sdata { 2, 0, 12, 15 };
    printBinary(std::cout, std::begin(sdata), std::end(sdata)) << "\n";
    // Output: 0000 0010 1100 1111
    int adata[] = { 0,1,2,13,14,15 };
    printBinary(std::cout, std::begin(adata), std::end(adata)) << "\n";
    // Output: 0000 0001 0010 1101 1110 1111
}

```

Listing 23.10 Pass your own algorithm iterators as arguments, not the container.

Specifically, `TYPE` is instantiated by `printBinary()` in this example with three different types:

- For `vdata.cbegin()`, `TYPE` is a `vector<int>::const_iterator`, as this is what the function `cbegin()` returns.
- For `std::begin(sdata)`, it is a `set<int>::iterator`, as that is the return type of `set<int>`. You would have gotten a `const_iterator` if you had declared `sdata` `const`.
- For `std::begin(adata)`, `TYP` will be a raw pointer `int*`. The loop in the `printBinary()` function, which looks like it specializes in iterators, also works on raw pointers—precisely because they are just a form of iterators.

I also did not use `auto` for the parameter here, did not specify a concept, and did not use template argument deduction. This is now C++14 compatible.

Writing your own functions on containers in this manner increases the flexibility of your program in the long run. Of course, this is not suitable everywhere. So long as you

only read the elements in the range, everything will definitely work. Here, we only let the loop run forward. Maybe you want to write an algorithm that accesses the elements of the range backward or even randomly. Then it may be that some containers do not cooperate with your function or behave undesirably. A `list` does not allow random access to its elements. With the choice of the right range concept, you can document that.

So long as you keep this in mind, you are definitely on the safe side for functions that only read. If you also need write access, you need to be a bit more cautious. When you describe an element with a new value, you must consider that, for example, a set wants to keep its elements sorted—and it will do so, causing the order of elements to change and thus the range as well.

23.2.2 Example: Information about Numbers

In the following example, you will not only see the use of a function template but also that there is a very interesting helper template, `numeric_limits<>`. With it, you can find out all sorts of useful internals and use them in your program. If you want to initialize a vector large enough to hold all possible values of unsigned short, you can do it like this:

```
const auto sz = std::numeric_limits<unsigned short>::max();
std::vector<string> dictionary(sz);
```

If the unsigned short type on your system has a width of 16 bits, then unsigned short variables can hold values from 0 to 65536. The `sz` variable takes this value, and the vector `dictionary` is initialized with this size.

A few examples of what else you can find out using `numeric_limits` are shown in [Listing 23.11](#).

```
// https://godbolt.org/z/r83Yeon7r
#include <iostream>                                // cout
#include <limits>                                    // numeric_limits
template<typename INT_TYP>                        // Template with type argument
void infos(const char* name) {
    using L = typename std::numeric_limits<INT_TYP>; // rename for brevity
    std::cout
        << name
        << " number of bits:" << L::digits          // Bits without sign bit
        << " sign:" << L::is_signed                 // stores sign?
        << " min:" << (long long)L::min()            // smallest possible value
        << " max:" << (long long)L::max()            // largest possible value
        << "\n";
}
```

```
int main() {
    infos<signed char>("char");           // smallest int type
    infos<short>("short");
    infos<int>("int");
    infos<long>("long");
    infos<long long>("long long");       // largest int type
}
```

Listing 23.11 How to find the range of integer types.

Although this example is only designed for integer types, there are also many information functions for the float and double floating-point types.

What Is “typename”?

So far, you have only seen `typename` for template parameters—for example, before functions in `template<typename T>`. In such cases, you can also use `class` instead: `template<class T>` means exactly the same thing.

Since C++20, you can also use the name of a concept instead of `typename` to restrict the type of a template parameter. If `infos` is supposed to work only for integer types, use the `std::integral` concept from the `<concepts>` header instead of `typename`.

Outside of angle brackets, a `typename` appears exactly when you are writing a template and a type you are using is itself a template type in which you are using one of your template parameters. You can see this in [Listing 23.11](#) at `typename std::numeric_limits<INT_TYP>`. Then you need the additional word `typename` before the name of the template type you are using. Otherwise, the compiler cannot know, before you assign a value/type to `INT_TYP`, whether `numeric_...<...>` is a variable, a type, or something else. You then have to tell it: “Hey, by the way, this is a type.”

23.3 A Class as a Function

So far, you can do two things with a function: you can define it, and you can call it. It becomes exciting and truly flexible only when you consider functions not just as a piece of outsourced program code but as objects like all other C++ objects. When you pass a function as a parameter, receive it as a return value, or simply store it in a variable for later use, you are using functions as objects.

I'll start with a very simple example. Say you want to write something that performs either an addition or a multiplication operation depending on a parameter. In the context of this chapter, I would like to call the thing that sometimes performs one operation and sometimes another a *procedure* to conceptually distinguish it from the variable function (addition or multiplication).

I don't even want to explicitly write down the crudest solution as a listing here. You could pass a `char` as a parameter, which performs an addition for a '+' sign and a multiplication for a '*' sign. You would then probably implement the functions themselves in a `switch` within the procedure. You already did that skillfully in [Chapter 8, Listing 8.18](#). By now, you would just be a bit more suspicious and use an `enum class` with enumeration elements for the types of functions.

I would much rather throw you straight into the deep end. The function itself is passed as a parameter. As a result, it gets the name of the parameter in the procedure, and you then call it normally with parentheses and parameters.

```
#include <functional> // function
int calculate(int a, int b, std::function<int(int,int)> binop) {
    return binop(a,b);
}
```

Listing 23.12 A function as a parameter.

The `binop` parameter has a somewhat peculiar type: `function<int(int,int)>`. This refers to any function that takes two `int` values as parameters and returns one. `function<...>` is a template, that much is clear. But the function must have the correct signature. And if you take a normal function declaration like

```
int plus(int arg1, int arg2);
```

and remove all names, `plus`, `arg1`, and `arg2`, you are left with `int(int,int)`. That is exactly the template argument for `function<...>`. You have determined the type of a function.

Here you need to take a deep breath first, because this is quite a chunk. Then let the actual invocation of the function parameter sink in:

```
binop(a, b);
```

The parameter `binop` behaves like a function. If you had defined `binop` as a global function, the call here would look the same.

23.3.1 Values for a “Function” Parameter

The procedure is thus complete. But what do you use as the third parameter for `calculate`? Try a function—of course, with the appropriate signature.

```
// https://godbolt.org/z/q719j41er
#include <functional> // function
#include <iostream>    // cout
```

```

int calculate(int a, int b, std::function<int(int,int)> binop) {
    return binop(a,b);
}
int plus(int a, int b) { return a+b; }
int times(int a, int b) { return a*b; }
int main() {
    std::cout << calculate(3, 4, plus) << "\n";    // Output: 7
    std::cout << calculate(3, 4, times) << "\n";    // Output: 12
}

```

Listing 23.13 Use the name of a suitable function as a function parameter.

That's all. The `plus` and `times` functions, instances of type `function<int(int, int)>`, are passed to the function `calculate()` as parameters.

It is important that the signatures of the functions match. For example, you cannot use the `negate` function. The compiler complains with an error:

```

int negate(int a) { return -a; }
// ...
std::cout << calculate(3, 4, negate) << "\n"; // ✗ wrong signature

```

23.3.2 C Function Pointer

In this section, I will tell you a bit about the history of C++. If your head is already spinning, you can skip it on the first read.

With my compiler, the error message is as follows:

```

could not convert 'negate' from 'int (*)(int)' to
'std::function<int(int,int)>'

```

So `negate` does not appear as `std::function<int(int)>` but as `int (*) (int)`. This is the actual type of a function when you pass it as a parameter. The `std::function<int(int)>` construct is just a C++ utility. The `int (*) (int)` notation comes from C times and is called a *function pointer* or *pointer*. The compiler implicitly converts a C-type function `int (*) (int)` to the corresponding C++-type `std::function<int(int)>`. Because the C notation looks very strange (at least subjectively to my eyes), I prefer the C++ notation wherever possible.

To be able to read what the compiler tells you, we need to decrypt the C type. The entire type of the function is `int (*) (int)`:

- The `(*)` in the middle means that it is some kind of function pointer.
- `int` at the beginning is the return type of the function signature.
- `(int)` in parentheses describes the parameter types of the function signature.

If you really want to, you can also write the corresponding parameter of `calculate()` in this C function pointer notation:

```
int calculate_c(int a, int b, int(*binop)(int,int)) {  
    return binop(a,b);  
}
```

Oh dear, and the name of the `binop` parameter is no longer behind the type, but right in the middle, next to the `*` and in the parenthesis for the function pointer (`*binop`).

I recommend that you stick with the C++ `function<>` notation, because I think it is much easier to read. And it has another advantage: the C++ type comes with all sorts of practical features that the C-function pointer does not have. Conversions and other actions are somewhat easier.

One last word on the term *function pointer*: In the C++ world, you imagine a function as some kind of object that you can simply pass around as a value with `plus` or `times`. That's all well and good, but in the C world, there were no complex objects. A function was always a concrete entity somewhere in memory. Therefore, if you want to pass the function as a parameter, you need to retrieve its *address* from memory using the address operator `&`. However, you don't need to do that here, because the compiler performs the necessary conversion implicitly and thus automatically.

```
// https://godbolt.org/z/c9Eh346cq  
#include <functional> // function  
#include <iostream>    // cout  
int calculate(int a, int b, std::function<int(int,int)> binop) {  
    return binop(a,b);  
}  
int plus(int a, int b) { return a+b; }  
int times(int a, int b) { return a*b; }  
int main() {  
    std::cout << calculate(3, 4, plus) << "\n";    // Value notation  
    std::cout << calculate(3, 4, times) << "\n";   // Value notation  
    std::cout << calculate(3, 4, &plus) << "\n";   // Pointer notation  
    std::cout << calculate(3, 4, &times) << "\n";  // Pointer notation  
}
```

Listing 23.14 It doesn't matter whether you use the address operator when calling or not.

Preview: Method Pointer

You have now seen how to make a function pointer from a free function: name the function without the call parenthesis. Optionally, you can also add a `&` for “address of”:

```
double func(int i);
auto funcPointer1 = &func;
auto funcPointer2 = func;
```

A special feature will be added with classes; see [Chapter 12](#). Classes have methods, and they are within the class. Otherwise, it works the same way, you just need to prefix the class name, but the `&` is no longer optional:

```
struct Class {
    double meth(int i);
};
auto methPointer = &Class::meth;
```

Because all methods receive an instance of the class as an implicit first parameter, `methPointer` now behaves like a function pointer that takes two parameters: the first of type `Class*`, the second of type `int`.

In the C++ world, where you can handle both values and addresses in function calls, you don't need to adjust your thinking in this case. Think of a function as a value if that concept suits you better, or think of it as a pointer. Personally, I prefer the value concept because it fits my idea of *functors*—real function objects, which we will now discuss.

23.3.3 The Somewhat Different Function

That you can call a function has become second nature to you by now—for example, in `return binop(a, b);`.

You think `binop` is a function? Maybe. Maybe not. It could be anything *callable*. Your natural counter question should be: “Well, what else is callable besides functions?” The answer: in C++, you call something by writing a pair of parentheses with parameters after the callable entity, and you get a result back; both the parameter list and the result can be empty. The following are callable:

- **Functions and methods**

You already know them: you define a header with return type, function name, call parameters, and the function body.

- **Functors or function objects**

A *functor* is a class that defines the `operator()`. Instances of this class then behave like a function.

- **Anonymous functions or lambdas**

If you want to pass an operation to an algorithm—for example, “How do I compare for sorting?”—you can pass the function body (the “How”) directly as a parameter without defining a function beforehand. This is why it is called an *anonymous function* or, in computer science jargon, a *lambda expression*.

Instances of a class with an `operator()` method behave like functions. You can call them as in the following listing.

```
// https://godbolt.org/z/xvfvG8hrd
#include <iostream>                                // cout
using std::cout;
class Increment {
    int amount_;
public:
    explicit Increment(int amount) : amount_{amount} {}
    int operator()(int value) const { // makes instances callable
        return value + amount_;
    }
    void clear() {
        amount_ = 0;
    }
};
int main() {
    Increment plusFour{4};                          // create instance
    Increment plusEleven{11};                       // another instance
    cout << plusFour(8) << "\n";                    // Output: 12
    int result = 2 * plusEleven(5) - 7;             // result is 25
    cout << plusEleven(result/5) << "\n";           // Output: 16
    cout << 3 * Increment{1}(7) << "\n";           // Output: 24
    Increment plusZero = plusEleven;
    plusZero.clear();                                // change state
    cout << plusZero(1) << "\n";                    // Output: 1
}
```

Listing 23.15 A class with “`operator()`” creates function objects.

If you only looked at the line with `plusFour(8)`, it would seem as if `plusFour` hides the function:

```
int plusFour(int value) {
    return value + 4;
}
```

And if you only looked at the line with `plusEleven(result/5)`, you might assume there is a function:

```
int plusEleven(int value) {
    return value + 11;
}
```

This could continue with +0, +1, +2, and so on—and you would wonder who would bother to write so many functions.

This is where function objects come into play. If a group of functions is very similar and only differs by one detail that can be represented by an internal state, then a class with data fields is the optimal solution.

Let's develop this slowly. If you only use ways you already know, you can write and use an increment class with normal methods as in the next listing.

```
// https://godbolt.org/z/MfKzd8bxa
#include <iostream>                                // cout
using std::cout;

class Add {
    int amount_;
public:
    explicit Add(int amount) : amount_{amount} {}
    int add(int value) const {                     // instead of operator()
        return value + amount_;
    }
    void clear() { amount_ = 0; }
};

int main() {
    Add plusFour{4};                               // create instance
    Add plusEleven{11};                            // another instance
    cout << plusFour.add(8) << "\n";               // Output: 12
    int result = 2 * plusEleven.add(5) - 7;         // result is 25
    cout << plusEleven.add(result/5) << "\n";       // Output: 16
    cout << 3 * Add{1}.add(7) << "\n";             // Output: 24
    Add plusZilch = plusEleven;
    plusZilch.clear();                             // change state
    cout << plusZilch.add(1) << "\n";             // Output: 1
}
```

Listing 23.16 A class with a normal method.

In fact, there are only two differences:

- Increment defines `int operator()(int value) const` as a method, while `Add` uses a normal `int add(int value) const` method for this.
- The call in the first case is `plusFour(8)`, in the second case `plusFour.add(8)`.

You see that the two classes actually only differ in that you save typing `.add(...)` when using the instances and instead write `(...)` directly—like a function call. And that's exactly why instances of classes with `operator()` are called *function objects*. The class with `operator()` itself is called a *functor*.

The Increment functor is indeed a rather complicated representative of its kind. In fact, the typical functor has neither an internal state like `amount_` nor modifiers like `clear()`. The typical functor is actually more like a different way to write a function—pardon me, a *callable object*.

Thus, function objects then integrate into the rest of the C++ structure:

- You can write class hierarchies with `operator()`.
- `operator()` can be virtual.
- You can overload `operator()`—that is, write multiple variants with different parameter types.
- You can represent internal states, as you have seen, and create variants via constructors.
- And, last but not least, you manipulate function objects like any other objects—for example, to put them in a container.

23.3.4 Practical Functors

Most functors are rather simple in design. You often use them as an interface between your own classes and the containers and algorithms of the standard library.

A `set<Type>` is always sorted according to `operator<(Type, Type)`. For example, you can create a list sorted by name with `set<Dwarf>` and the appropriate `operator<`.

```
// https://godbolt.org/z/3ch51ov87
#include <set>
#include <string>
#include <iostream> // cout
using std::string; using std::set; using std::cout;
struct Dwarf {
    string name_;
    unsigned year_;
};
bool operator<(const Dwarf& a, const Dwarf& b) {
    return a.name_ < b.name_;
}
int main() {
    set dwarves{ Dwarf{"Balin", 2763}, Dwarf{"Dwalin", 2772},
                Dwarf{"Oin", 2774}, Dwarf{"Gloin", 2783}, Dwarf{"Thorin", 2746},
                Dwarf{"Fili", 2859}, Dwarf{"Kili", 2864} };
    for(const auto& z : dwarves) // sorted output: "Balin" to "Thorin"
        cout << z.name_ << " ";
    cout << "\n";
}
```

Listing 23.17 With “`operator<`” as a function, you can only implement one sort order.

By the way: since C++20, instead of `operator<...`, you simply write `auto operator<=>(...)` = default.

This way, you get the output sorted by name:

Balin Dwalin Fili Gloin Kili Oin Thorin

Assume that fulfilled 95% of your program's purpose, but in 5% of the cases you want to output by age. What can you do? You can only define one `operator<` (except with multiple namespaces).

For this purpose, `set` offers a second template parameter. Provide `set` with a functor that has the appropriate comparison operation. This will then be used instead of `operator<` for `set` sorting. Insert the following functor into [Listing 23.17](#).

```
struct ByYear { // implements less-than by Dwarf::year_
    bool operator()(const Dwarf& a, const Dwarf& b) const {
        return a.year_ < b.year_;
    }
};
```

Listing 23.18 A no-frills functor with great utility.

Then you can achieve sorting by the birth year of the dwarves with the following addition in `main`:

```
// https://godbolt.org/z/PebG9bn9K
set<Dwarf, ByYear> dwarves2{begin(dwarves), end(dwarves)};
for(const auto& z : dwarves2) // differently sorted output
    cout << z.year_ << " ";
```

`dwarves2` stores a copy of all elements from `dwarves`, because the constructor that takes two iterators copies all elements. This time, however, the elements are sorted using the `ByYear` functor. Instead of calling `operator<` for each comparison, the following is now used:

```
bool MyYear::operator()(const Dwarf&, const Dwarf&) const;
```

Thus, you get the following output:

2746 2763 2772 2774 2783 2859 2864

As you can see, you cannot use the C++17 feature to omit the `Dwarf` template parameter, because you have to provide the second parameter with `ByYear`; the compiler cannot deduce it from the constructor arguments. But `set` has a constructor that can do exactly that. Here, I pass `ByYear{}` as a constructor argument, and thus the compiler can deduce the template parameters:

```
set dwarves2{begin(dwarves), end(dwarves), ByYear{}}; // ⚡ not deducible
```


23.3.5 Algorithms with Functors

Functors are not only interesting for the classes of standard containers. Many helper functions that can be found in the standard library can be modified in behavior with a callable object.

For example, you can sort a vector using the same mechanism with `operator<` if you use the `sort()` function from the `<algorithm>` header.

```
// https://godbolt.org/z/YKzsW49G8
#include <vector>
#include <algorithm> // sort
// Definitions and further includes as before
int main() {
    vector dwarves{ // initialize as before
/* sort */
    std::sort(begin(dwarves), end(dwarves));
// output as before ...
```

Listing 23.19 Many algorithms in the standard library also work with “operator<”.

`sort()` uses `operator<` by default for dwarf, and you get a list sorted by name from “Balin” to “Thorin”.

However, you can also call `std::sort()` with an additional parameter: a functor object. Here, you do not need a template parameter but can instantiate `ByYear` for this purpose:

```
ByYear byYear{};
std::sort(begin(dwarves), end(dwarves), byYear);
```

Or shorter—with the instance created directly on the spot as a temp-value without a name:

```
std::sort(begin(dwarves), end(dwarves)), ByYear{});
```

You will then get a sorted vector from 2746 to 2864.

Since C++20, you can write `std::ranges::sort(dwarves, ...)` and no longer need to worry about iterators.

23.3.6 Anonymous Functions: a.k.a. Lambda Expressions

You might say now that it is quite a lot of effort to define a class that has a specific method, then create an instance, and finally pass it to `sort()`—just to communicate the `a.year_ < b.year_` operation as the “how.”

You are not alone in this. Therefore, there is a way to define a functor directly on the spot where it is needed, as the third parameter in `sort()`. You write the function body of `operator()` directly at the appropriate place:

```
std::ranges::sort(dwarves,
    [](const Dwarf& a, const Dwarf& b) { return a.year_ < b.year_; }
);
```

As you can see, you also need to inform the compiler with `(const Dwarf&a, const Dwarf&b)` about what the parameters of `operator()` actually are. The construct is introduced with `[]`. In the past, before C++20 ranges, you would write `std::sort(begin(dwarves), end(dwarves), ...)`.

You have just created an *anonymous function*. Except for the name, you have all the elements of a function definition.

What is still missing—and you have surely noticed this—is the return type. The compiler can infer it in most cases. If you want (or need) to specify it explicitly for an anonymous function, then use the trailing `->` notation:

```
std::ranges::sort(dwarves,
    [](const Dwarf& a, const Dwarf& b) -> bool { return a.year_ < b.year_; }
);
```

You can translate any lambda expression directly into a functor class with a name. The anonymous function

```
std::ranges::sort(dwarves,
    [](const Dwarf& a, const Dwarf& b) -> bool { return a.year_ < b.year_; }
);
```

becomes the following:

```
struct F {
    bool operator()(const Dwarf& a, const Dwarf& b) const
    { return a.year_ < b.year_; }
};
std::sort(begin(dwarves), end(dwarves), F{});
```

Perhaps this transformation will help you understand.

Lambdas with External Access

Within the statement block of the anonymous function, you cannot access surrounding variables. Why would you want to do something like that? A typical use case is, for example, a variable that, unlike the arguments of the `const Dwarf& a` and `const Dwarf& b` lambda, does not change with each call. You might only change the general behavior of the comparison.

Let's incorporate a Boolean switch that indicates whether we want to sort forward or backward. You can implement the reversal of the order by comparing with greater than > instead of less than <.

```
// https://godbolt.org/z/eh6jnjk9z
#include <vector>
#include <string>
#include <algorithm> // sort
#include <iostream> // cout
using std::string; using std::vector; using std::cout;
// as before
int main() {
    vector dwarves{           // as before
/* sort */
        bool backwards = true; // or false. Variable outside the lambda
        std::ranges::sort(dwarves,
            [backwards](const Dwarf& a, const Dwarf& b) {
                if(backwards)
                    return a.name_ > b.name_;
                else
                    return a.name_ < b.name_;
            }
        );
/* output */
    for(const auto& z : dwarves) // backwards: "Thorin" to "Balin"
        cout << z.name_ << " ";
    cout << "\n";
}
```

Listing 23.20 With the capture clause, you can access external variables in the lambda.

Within the square brackets of the anonymous function, you now see a *capture clause*. Without this announcement that you want to access `backwards` in the lambda, the compiler would respond to the use of `if(backwards)...` with an error.

You must consider the following points:

- The declarations in the capture clause are copied as values. This happens once—namely, during the initialization of the lambda.
- Within the function body, the copy is `const`. An assignment like `backwards = false` within the lambda would therefore not work.

If you do not want to copy, you also have the option to access the outer variable by reference. Then precede the name with a reference symbol `&` in the capture clause. This also has the effect that you can modify the variable. You then modify—as is usual with references—the original variable.

The following program, for example, counts the number of swaps needed during sorting.

```
// https://godbolt.org/z/j7qoMdG3n
#include <vector>
#include <string>
#include <algorithm> // sort
#include <iostream> // cout
using std::string; using std::vector; using std::cout;
// as before
int main() {
    vector dwarfs{           // as before
/* sort */
    bool backwards = true;    // or false. Variable outside the lambda
    unsigned rightway = 0;    // counts <
    unsigned wrongway = 0;    // counts >
    std::ranges::sort(dwarves,
        [backwards,&rightway,&wrongway](const Dwarf& a, const Dwarf& b) {
            bool result = backwards
                ? a.name_ > b.name_
                : a.name_ < b.name_;
            if(result==false) ++wrongway; else ++rightway;
            return result;
        }
    );
/* output */
    cout << "Wrongway:" << wrongway << " Rightway: " << rightway << "\n";
    for(const auto& z : dwarves) // backwards: "Thorin" to "Balin"
        cout << z.name_ << " ";
    cout << "\n";
}
```

Listing 23.21 The capture clause can also contain references.

This shows that sorting the seven dwarfs requires 17 comparisons. Among these, six comparisons were in “incorrect order” and eleven were in “correct order.”

```
Backwards:6 Forwards: 11
Thorin Oin Kili Gloin Fili Dwalin Balin
```

You should only use references in the access clause when absolutely necessary. As always, references carry certain risks. With anonymous functions, it is quite common to send them on a journey—that is, pass them as parameters to other functions, return them as results, and so on. It may happen that the lambda then “lives” longer than the referenced variable. However, if you accessed a copy, there is no danger.

Mutable Lambdas

In [Chapter 13, Section 13.10.10](#), I promised to explain `mutable` for lambdas, and I will do so here.

With `mutable`, you allow changes to capture variables that were copied into the lambda. To do this, place the word `mutable` after the lambda.

```
// https://godbolt.org/z/zMWO69974
#include <iostream>
int main() {
    int count = 0;
    auto plus1 = [count](int x) mutable { // count as a copy
        std::cout << ++count; return x+1;
    };
    for(int i=0; i<5; ++i) {
        plus1(i);
    }
    std::cout << "\n";
    // Output: 12345
}
```

Listing 23.22 “mutable” makes value captures in lambdas mutable.

Even `mutable` should be used with caution. Maybe you want a reference capture on count. If so, you can change that without `mutable` as well.

```
// https://godbolt.org/z/8GxPMEc5o
#include <iostream>
int main() {
    int count = 0;
    auto plus1 = [&count](int x) { // count as reference
        ++count; return x+1;
    };
    for(int i=0; i<5; ++i) { plus1(i); }
    std::cout << "plus1 was called " << count << " times\n";
    // Output: plus1 was called 5 times
}
```

Listing 23.23 If possible, better without `mutable`.

However, if the lambda is passed as a return value or as a parameter, then a reference can be dangerous because the referenced variable might no longer exist. `mutable` can help here in exceptional cases.

For Your Convenience

Now the only thing left to tell you is that if you have a fully packed capture clause, you can also abbreviate it:

- With `[a=b]`, you initialize the lambda variable `a` with the outer expression `b`.
- `[&r=s]` initializes the reference `r` for the lambda with the outer expression `s`.
- With `[=]`, you can access all visible variables by value (copy). Avoid this especially in member functions, because currently `*this` is also copied (i.e., the entire object). Copying `*this` is deprecated as of C++20.
- `[&]` allows you to access all variables by reference.
- `[this]` and `[*this]` allow lambdas in member functions to access the entire instance as a reference or copy, respectively (the latter from C++17 onward). Since `[=]` will soon no longer copy `*this`, it will be necessary to explicitly write `[=,*this]`.
- `[...]` expands in templates with a variable number of arguments—and is combinable with `&` (from C++20 onward).
- All these elements can be combined, separated by commas.
- You can specify with `[&,backwards]` that you want to access all variables by reference except for `backwards`, which is available to you as a copy.
- Conversely, `[=,&wrong,&right]` provides you with all variables as copies, with only `wrong` and `right` as references.

If the name of the access variable seems unsuitable for use within the function body, you can also declare a new variable in the list of access variables, which is defined with an *access variable initialization clause*:

```
bool backwards = true;    // or false; variable outside the lambda
unsigned correct = 0;     // counts < with
unsigned incorrect = 0;   // counts > with
std::sort(begin(dwarves), end(dwarves),
    [dir=backwards,&cf=incorrect,&cr=correct](const Dwarf&a, const Dwarf&b) {
        bool result = dir ? a.name_ > b.name_ : a.name_ < b.name_;
        if(result==false) ++cf; else ++cr;
        return result;
    });
```

You will probably need this rather rarely.

More often, you might find yourself using the ability to let the compiler automatically determine the parameter types of the anonymous function. Then, instead of the specific type—in our case, `Dwarf`—simply use `auto`. The compiler can usually determine the necessary type from the context:

```
std::ranges::sort(dwarves,  
    [](const auto &a, const auto &b) { return a.year_ > b.year_; }  
);
```

Otherwise, the usual rules for type inference with `auto` in conjunction with `const` and `&` applies. If you want to receive the parameter as a value (copy), just write `auto`. If you want it as a constant reference, use `const auto&`.

23.3.7 Template Functions without “template”, but with “auto”

And now you can surely put two and two together: a lambda is a kind of function, `auto` determines the type of the parameters, and I can assign a lambda to a variable, where `auto` also takes care of the annoying return type. So why shouldn't I combine everything and get a compact, readable (because there are no angle brackets) definition of a function template?

```
// https://godbolt.org/z/xnbvEx5Kc  
#include <iostream>  
#include <string>  
using std::cout; using std::string; using namespace std::literals;  
auto min2 = [](const auto &a, const auto &b) {  
    return a < b ? a : b;  
};  
auto min3 = [](const auto &a, const auto &b, const auto &c) {  
    return min2(a, min2(b,c));  
};  
int main() {  
    cout << min3( 3, 7, 2 ) << '\n';           // Output: 2  
    cout << min3( 8.11, 113.2, -3.1 ) << '\n'; // Output: -3.1  
    cout << min3( "Zoo"s, "Ape"s, "Mule"s ) << '\n'; // Output: Ape  
}
```

You saw in [Section 23.1.3](#), in the “C++20: Abbreviated Function Template” box, that this use of `auto` has also been applicable to functions since C++20. And also introduced with C++20 is the ability to write a true template lambda:

```
// https://godbolt.org/z/nenW8aM81  
auto min2 = [<typename T>(const T &a, const T &b) {  
    return a < b ? a : b;  
};  
auto min3 = [<typename T>(const T &a, const T &b, const T &c) {  
    return min2(a, min2(b,c));  
};
```

With this feature, you enforce that the parameters must have the same type.

23.4 C++ Concepts

Since C++20, there is the possibility to restrict the types you use as template parameters. For this, *concepts* were introduced. These define language within C++ that allows you to make assertions about template parameters. When you then instantiate (i.e., use) a template, these assertions are checked and acknowledged with a clearer error message than would be possible without a concept.

The standard library comes with a lot of concepts, many of them in the `<concepts>` header, but also in `<iterators>`, `<ranges>`, and so on.

23.4.1 How to Read Concepts

When you look at a function header with a concept, you see something like the following listing.

```
// https://godbolt.org/z/jP75nY4EW
#include <concepts> // integral
#include <iostream>
using namespace std;
// Concept explicitly with requires
template<typename T> requires integral<T>
auto add_1(T val) { return val+1; }
// abbreviated concept
template<integral T>
auto add_2(T val) { return val+2; }
// abbreviated function template with concept
auto add_3(integral auto val) { return val+3; }
// Ad-hoc requirements for function template
auto add_4(auto val) requires integral<decltype(val)>
{ return val+4; }
int main() {
    cout << add_1(1) << '\n';           // Output: 2
    cout << add_2(1) << '\n';           // Output: 3
    cout << add_3(1) << '\n';           // Output: 4
    cout << add_4(1) << '\n';           // Output: 5
    cout << add_3("text") << '\n';     // ✗ Error
    integral auto val = add_1(99);      // also for auto variables
    cout << val << '\n';               // Output: 100
}
```

Listing 23.24 Concepts can be written in different ways.

The functions `add_1`, `add_2`, `add_3`, and `add_4` are essentially the same, just using different notations:

- `add_1` uses the verbose syntax with `requires` after the template header. Here, the type `T` has a name that you can refer to in the `requires` part.
- `add_2` uses a particularly compact shorthand in the template header that utilizes a previously defined concept.
- `add_3` uses the abbreviated function template with `auto` parameter instead of the template header. You hardly notice that you are defining a template.
- `add_4` compensates for the disadvantage of `add_3`, which is that without a template header, you don't have a name for the template parameter. However, you can determine the type of the parameter with `decltype` and then restrict it even after the function header with `requires`.

The added value is apparent when you try to call the function with a non-integral. The compiler reports to me at `add_3("text")` (excerpt):

```
error: no matching function for call to 'add_3(const char [5])'
template argument deduction/substitution failed:
required for the satisfaction of 'integral<const char*>'
```

This error message states directly and precisely that `const char*` is not integral and therefore cannot be used.

In the headers and documentation of the standard library, many parameters and return types are now described using concepts. This way, you know what parameters the template functions and classes expect.

In the case of `integral`, it's simple: for type `T`, the trait `is_integral S<T>::value` must be defined as `true`. According to the standard, this is the case for `bool`, `char`, `charNW_t`, `wchar_t`, `short`, `int`, `long`, `long long`, and their unsigned versions (the compiler implementation may offer more).

But not only the error messages get better. Concepts can also distinguish between multiple overloads of a function that would otherwise be indistinguishable. An API could, for example, offer the same function overloaded once for `integral` and once for `floating_point`.

```
// https://godbolt.org/z/Kxrz5ETh6
#include <concepts> // integral, floating_point
#include <iostream>
using namespace std;
int64_t trunc_to_10(integral auto val) { return (val/10)*10; }
int64_t trunc_to_10(floating_point auto val) { return int64_t(val/10)*10; }
int main() {
```

```
cout << trunc_to_10(144) << '\n';    // Output: 140
cout << trunc_to_10(122.2) << '\n';  // Output: 120
}
```

Listing 23.25 With concepts, you can restrict individual overloads.

The `integral` implementation covers all data types from `char` to `long long`, while the `floating_point` implementation covers everything from `bfloat16` to `long double`.

Because the implementations for `integral` and `floating_point` are slightly different, concepts provide an easy way to separate the two implementations. Without concepts, you would either have to provide many overloads from `char` to `long double` or use type traits, which are not as easy to read.

The `requires` shown so far always introduce a `requires` clause. I have always referred to an existing concept. However, you can also encounter a `requires` expression to formulate a condition ad hoc:

```
template<typename T>
    requires requires(T p) { *p; }
auto max_deref_value(T a, T b) { return *a > *b ? *a : *b; }
```

The first `requires` is the introduction of the `requires` clause; the second is the introduction of the `requires` expression for `T p` when you cannot refer to a named concept.

You can find more useful concepts of the standard library in [Table 23.1](#). There are many more, especially in the `<iterator>` and `<ranges>` headers.

Many concepts of the standard library build on each other, creating a kind of hierarchy. For example, `regular` includes the `equality_comparable` and `semiregular` concepts—with `semiregular` in turn requiring `copyable`, which then requires `movable`.

Concept	Description
<code>same_as</code>	T and U are the same type
<code>derived_from</code>	T is derived from U
<code>convertible_to</code>	T is implicitly convertible to U
<code>integral</code>	Integral type
<code>signed_integral</code>	Signed integral type
<code>unsigned_integral</code>	Unsigned integral type
<code>floating_point</code>	Floating-point type
<code>assignable_from</code>	T is assignable from U

Table 23.1 Some useful concepts from the standard library.

Concept	Description
swappable	swap() is applicable on T
destructible	Can be removed without throwing an exception
constructible_from	Constructor can be called with Args...
default_initializable	Has default constructor
move_constructible	Has move constructor
copy_constructible	Has copy constructor
copyable	Copyable
movable	Can be moved
equality_comparable	Can be compared with ==
totally_ordered	Can be compared with <
three_way_comparable	Can be compared with <=> (from <compare>)
semiregular	Copy, move, and default constructor
regular	Semiregular and ==
invocable	F can be called with Args...
predicate	F can be called with Args... and returns bool
relation	F can be called with T and returns bool

Table 23.1 Some useful concepts from the standard library. (Cont.)

23.4.2 How to Use Concepts

In the previous section, you saw some examples of self-written functions that work with existing concepts. When you write functions and classes with concepts yourself, you have a few more tools at your disposal.

The easiest way is to write a concept name instead of `typename` in the angle brackets `<>` of the template header or before the `auto` of an abbreviated function template:

```
// https://godbolt.org/z/eaGre4bqr
auto min2(const std::integral auto &a, const std::integral auto &b) {
    return a<b ? a : b;
};

auto min3 = [<std::integral T>](const T &a, const T &b, const T &c) {
    return min2(a, min2(b,c));
};
```

```
int main() {
    cout << min3( 3, 7, 2 ) << '\n';
    cout << min3( 8.11, 113.2, -3.1 ) << '\n'; // ✗ Error: not an integral type
}
```

Here we have restricted the arguments with `integral` so that `min3(8.11,...)` no longer compiles. This works with functions like `min2()` as well as with lambdas like `min3()`.

But you have even more options. First, you have already learned about the `requires` keyword. With it, you can not only specify a single concept but also combine multiple ones and add further conditions.

```
// https://godbolt.org/z/fxoo4h9Gj
#include <concepts>
#include <iostream>
#include <string>
using namespace std; using namespace std::literals;

string mk_string(integral auto val) { return to_string(val); }
string mk_string(string val) { return '['+val+']'; }
template<typename T>
    requires copyable<T> && // Type requirement
    requires (T t) { mk_string(t) + mk_string(t); } && // simple requirement
    requires (T t) { // composite requirement
        {mk_string(t)} -> same_as<string>; // valid expression must meet condition
    }
string dbl_quote(const T& val) {
    T val2{val}; // Create a copy (for demonstration purposes only)
    return '"' + mk_string(val) + mk_string(val2) + '"';
}
int main() {
    cout << dbl_quote(10) << '\n'; // Output: "1010"
    cout << dbl_quote("ab"s) << '\n'; // Output: "[ab][ab]"
    cout << dbl_quote(3.14) << '\n'; // ✗ Error: no suitable mk_string overload
}
```

Listing 23.26 Specify additional conditions with `requires`.

Assume there are overloads for `mk_string()` defined only on `integral` and on `string`, but not on `float`. Now we want to write a function `dbl_quote()` that copies a value and then converts it to a string using `mk_string()`. We express this by first introducing the template with `template<typename T>`, thus defining the placeholder `T`.

For this `T`, we then specify some `requires`, which we combine with `&&`. The first `requires` is the introduction of the entire `requires` clause. The subsequent `requires` each introduce `requires` expressions within the clause. OR combinations with `||` are also possible, but much rarer.

The AND combination consists of the individual requirements that must be met. There are three types:

- **Type requirement**

In the example, it is `copyable<T>`. The type `T` is used in another concept that must be true.

- **Simple requirement**

The example states that the `mk_string(t) + mk_string(t)` expression must be valid for a `t` of type `T`. The expression is not *executed*; it is only checked whether the expression compiles based on the types.

- **Composite requirement**

Beyond the simple requirement, the arrow `->` indicates a type requirement for the expression.

The simple and composite requirements should each end with a semicolon `;`.

23.4.3 How to Write Concepts

Now, it would be tedious to repeat the entire list of requirements every time you write a function. Therefore, you can define recurring concepts with the `concept` keyword.

```
// https://godbolt.org/z/Tcxnr6jTM
template<typename T>
    concept dbl_quotable = copyable<T> &&
        requires(T t) { mk_string(t) + mk_string(t); } &&
        requires(T t) {
            {mk_string(t)} -> same_as<string>;
        };
string dbl_quote(const dbl_quotable auto& val) {
    auto val2{val}; // Create a copy (for demonstration purposes only)
    return '"' + mk_string(val) + mk_string(val2) + '"';
}
```

Listing 23.27 Define recurring requirements with `concept`.

What you previously wrote directly in front of the function with an `&&` chain of `requires`, you now define with `concept` and give it a name. You can then use this name in later templates. In the example, I implemented this with the `auto` of an abbreviated function template.

Finally, I want to mention that you can also use concepts together with `using`. This way, you can also restrict a type alias with a concept:

```
// https://godbolt.org/z/9qsGW6M8Y
template<std::ranges::range T>           // a range over type T
using ValueTypeOfRange = std::ranges::range_value_t<T>;
// or
template<typename T>
requires std::ranges::range<T>          // a range over type T
using ValueTypeOfRange = std::ranges::range_value_t<T>;
// results in:
ValueTypeOfRange<std::vector<int>>> x;   // x is int
ValueTypeOfRange<std::string> y;         // y is char
ValueTypeOfRange<std::list<double>>> z;  // z is double
ValueTypeOfRange<int> w;                 // ✗ int is not a range
```

Even within `constexpr` if you can use concepts. The following listing shows a minimal example of it.

```
if constexpr std::integral<T> {
    // T is an integral type
} else if constexpr std::floating_point<T> {
    // T is a floating-point type
} else {
    // T is neither
}
```

Listing 23.28 This is how you use concepts in a `constexpr` if.

Here, `if constexpr std::integral<T>` checks whether `T` is an integral type, and then `if constexpr std::floating_point<T>` checks whether `T` is a floating-point type. A different example can be found in [Chapter 25, Listing 25.6](#), where also two `if constexpr` check for concept requirements.

23.4.4 Semantic Constraints

So far, I have mainly shown you how to constrain templates using concepts that work based on *syntactic errors*. This means that if something would not compile (or would result in nonsense, as in the case of pointer comparisons), you constrain it with a concept.

However, the standard library also uses concepts to check the semantics of templates. This means that even if it would technically work, you can use traits or other agreements to change the semantics so that concepts notice this and enforce a constraint.

One example is the `ranges::sized_range` concept. It requires that it must be possible to calculate the size of a range in $O(1)$. A container that offers `size()` usually fulfills this—at least, it is assumed. However, if you write a `Pile` container yourself that has a `size()` method which *does not* work in $O(1)$, then the `sized_range` concept should not be fulfilled for this container. This is a semantic constraint.

The `ranges::sized_range` concept therefore also checks whether the `disable_sized_range<Pile>=true` expression is defined—or, more precisely:

```
constexpr bool std::ranges::disable_sized_range<Pile> = true;
```

So you can use your own definition to communicate the semantics of your container, and the `sized_range` semantic concept will notice it.

23.4.5 Interim Recap

Concepts are a way to define requirements for types and expressions. Here are some additional tips:

- Document your concepts.
- Prefer named concepts. A name alone is already a piece of documentation, plus it promotes reuse. As a rule of thumb: Are you about to write `<typename ...>` or `<class ...>`? Chances are high that it is better to write `<SomeConcept ...>`.
- Use the standard library concepts.
- Utilize concepts to improve error messages.
- Apply concepts to local variables that you have declared with `auto`.
- Do not make your concepts too complex.
- Always format concepts consistently. Follow the conventions of your work environment.

23.5 Template Classes

The template technique, as you have just learned with function templates, is not limited to functions but can also be used with classes. To be precise, C++ would not be as popular and beloved as it is today without class templates. The standard library defines many class templates, of which you have already used `std::vector` or `std::string`. With `std::vector`, you specified the type in the angle brackets. `std::string`, on the other hand, is an instantiation of the `char` type and is internally defined as follows:

```
using string = basic_string<char>;
```

In addition to the standard container classes like `std::vector`, `std::list`, and so on, the stream classes for input/output (see [Chapter 26](#)) are also mostly implemented as instantiations of `char`.

23.5.1 Implementing Class Templates

To start, I want to show you how you can create a very primitive container (a container class) that can hold (almost) any type. The introduction of a class template here again uses the following statement:

```
template <typename T>
```

Again, `T` stands for the formal data type, which the compiler replaces with the desired type during instantiation—that is, the generation of machine code. The name `T` can again be any valid identifier; `T` is a very common usage here. You can also use `<class T>` instead of `<typename T>`. As with function templates, it makes no difference at this point what you use.

Since C++20, you can use a concept instead of `typename` here. This would be the shortened and preferred notation. Instead, after this line, a requirement for the type `T` can also be specified with `requires`.

The following listing shows the definition of a complete class template.

```
// https://godbolt.org/z/jqzTMxYKv
template <std::copyable T>      // C++20 concept
class MyContainer {
    T data_;
public:
    void setData(const T& d) { data_ = d; }
    T getData() const { return data_; }
};
```

Listing 23.29 A class template takes a type as a formal parameter.

At the beginning, I defined a `MyContainer` class template, which can now almost be defined like `std::vector<T>`; of course, many functionalities are still missing, but this already serves as a theoretical comparison. I have sensibly restricted `T` with the copyable C++20 concept. However, you can also simply write `typename` or `class`. If you were to create an object like `MyContainer<int>` now, the compiler would replace all *formal data types* (here `T`) with `int` and generate machine code accordingly. You will learn more about instantiating objects as class templates shortly.

23.5.2 Implementing Methods of Class Templates

In the recently created example of the `MyContainer<T>` class template, you could see that it is quite possible to set the formal data type in the parameters like `setData` or the return value like `getData` of methods.

Prefer Passing Arguments of the Templated Type as References or Pointers

Here, the recommendation is still missing that values, as in the example with `setData(const T&)`, should be passed as references (or pointers) and not as copies. Because if the class template is used with objects, quite a lot of data could accumulate, which would then unnecessarily be placed on the stack and removed again. However, with template functions and template classes, you must consider more carefully than usual whether you want to pass parameters by value or by reference.

In practice, you will only write the definition of the method inline within the class template definition if it is really a very short function body. You already know how to define a method outside of a class, but in the case of a method of a class template, it is somewhat more specific, and therefore it needs to be addressed separately here. The syntax for a method of a class template with formal data types is as follows:

```
template <typename T>
void ClassName<T>::method( parameter ) { ... }
```

With reference to our `MyContainer` class template, this definition of the method outside the class template definition would then be as shown in the following listing.

```
// https://godbolt.org/z/4fk5WdoG8
template <std::copyable T>
class MyContainer {
    T data_;
public:
    void setData(const T& d);
    T getData() const;
};
template <std::copyable T>
void MyContainer<T>::setData(const T& d) {
    data_ = d;
}
template <std::copyable T>
T MyContainer<T>::getData() const {
    return data_;
}
```

Listing 23.30 Method definitions outside the body of a class template are syntactically somewhat more complex.

It is important that for methods of class templates, you also specify the formal data type in angle brackets (here `<T>`) after the class name, as seen in `MyContainer<T>::...`. A specification like `MyContainer<T>` corresponds to specifying a complete type. An “empty” `MyContainer` would merely be a template name.

Preliminary Definition

As with function templates, the definition of a method is only a preliminary definition; the type is still incomplete as it only contains a formal data type. The actual methods are generated only when you instantiate an object of the class template—that is, use it for the first time, meaning that you specify it with a concrete type.

Overriding (Specializing) Methods

As with function templates, you can override (*specialize*) methods of class templates as needed with `template<>` and the desired data type. Instead of `MyContainer<T>`, you must then specify the actual data type for the formal data type `T`. You can implement such a specialization for `std::string` for the `setData()` and `getData()` methods as shown in the following listing.

```
// https://godbolt.org/z/M6earoWMn
#include <iostream>
#include <string>
#include <concepts>    // copyable, C++20
template <std::copyable T>
class MyContainer {
    T data_;
public:
    void setData(const T& d) { data_ = d; } // general case
    T getData() const { return data_; }    // general case
};
template<>                                // Specialization
void MyContainer<std::string>::setData(const std::string& d) {
    data_ = "[" + d + "]";
}
int main() {
    MyContainer<std::string> mcString;
    mcString.setData("History");
    std::cout << mcString.getData() << '\n'; // Output: [History]
    MyContainer<int> mcInt;
    mcInt.setData(5);
    std::cout << mcInt.getData() << '\n';    // Output: 5
}
```

Listing 23.31 Specializing class template methods like function templates.

Here you see a specialization of `setData()` for `MyContainer<std::string>`, which is used and generated when you instantiate a `MyContainer<std::string>` object, as briefly shown in `main()`. The general case remains and applies to all formal type parameters that have not received a specialization. Therefore, the output for `string` is with square brackets—as programmed in the specialization—but without for `int`—as programmed in the general case.

`std::copyable` is a C++20 concept. Without concepts, you write `typename`.

23.5.3 Creating Objects from Class Templates

I will now go into more detail about creating instances from a class template. As with function templates, class templates are instantiated with the corresponding type only upon first use. You have already used instantiating class templates multiple times in this book, such as `vector<int>`. In addition to the class name, you must specify the type for the formal data type between the angle brackets, from which the compiler should generate a class.

The following listing shows a few classic application examples of the minimal `MyContainer<T>` example class.

```
// https://godbolt.org/z/9vP3o68bW
#include <iostream>
#include <string>
#include <concepts>    // copyable, C++20
template <std::copyable T>
class MyContainer {
    T data_;
public:
    void setData(const T& d) { data_ = d; }
    T getData() const { return data_; }
};

class IntValue {
    int val_;
public:
    explicit IntValue(int val=0) : val_(val) {}
    int getInt() const { return val_; }
};

int main() {
    // C-array with three MyContainer<double> instances
    MyContainer<double> dcont[3];
    dcont[0].setData(123.123);
    dcont[1].setData(234.234);
    std::cout << dcont[0].getData() << std::endl;
```

```

    std::cout << dcont[1].getData() << std::endl;
    // custom data type as formal parameter
    IntValue ival{100'000};
    MyContainer<IntValue> scont;
    scont.setData(ival);
    std::cout << scont.getData().getInt() << std::endl;
    // string as formal parameter
    std::string str("Text");
    MyContainer<std::string> strCont;
    strCont.setData(str);
    std::cout << strCont.getData() << std::endl;
}

```

Listing 23.32 This is how you create instances from a template class.

When declaring `MyContainer<double> dcont[3]`, the compiler generates a class for `MyContainer<double>` and then creates three instances of this class for the C-array. The compiler replaces the formal data type `T` of the class template with the concrete `double` type. With `dcont[...].setData(...)`, I set values for some of the container instances and output them.

This also works with custom classes, as I demonstrate with the `MyContainer<IntValue> scont` instantiation. Here, the compiler will generate a class for `MyContainer<IntValue>`. The compiler replaces the formal data type `T` with the `IntValue` class.

In the last instantiation, `MyContainer<std::string> strCont`, another form of the class is generated by the compiler. Here, it substitutes the formal data type with `std::string`.

Double Error Checking

Another advantage when instantiating templates in general is that the compiler has to check for errors twice: once when parsing the template definition and again when instantiating the template with the specific type.

Class Template Argument Deduction (CTAD)

Since C++17, the compiler can deduce the type(s) of the parameters for the template class based on the types of the constructor arguments itself (via class template argument deduction [CTAD], a.k.a. template parameter deduction). Up to and including C++14, you always had to explicitly list the type parameters. You have already seen this for function templates, specifically in [Section 23.1.2](#). In the following listing, it is now the classes that are templates.

```

// https://godbolt.org/z/YaEW5aTra
#include <string>
#include <vector>

```

```
#include <set>
#include <tuple>
#include <memory> // shared_ptr
template <typename T>
class MyStuff {
    T data_;
public:
    MyStuff() : data_{} {} // default constructor
    MyStuff(const T& d) : data_{d} {} // copy constructor
    T getStuff() const { return data_; }
};
class IntValue {
    int val_;
public:
    explicit IntValue(int val=0) : val_(val) {}
    int getInt() const { return val_; }
};
int main() {
    MyStuff intStuff(12); // becomes MyStuff<int>
    std::vector vs{ 1,2,3,4 }; // becomes vector<int>
    MyStuff ivalStuff{ IntValue{33} }; // becomes MyStuff<IntValue>
    std::tuple tpl{ 23, 'a' }; // becomes tuple<int,char>
    std::set twoThree (vs.begin()+1, vs.end()-1); // becomes set<int>
    // The compiler cannot deduce:
    MyStuff<double> dblStuff; // no constructor argument
    std::vector<char> vcs(10); // size 10, but of what type?
    std::shared_ptr<int> ptr{new int{88}}; // no rule for int*
}
```

Listing 23.33 Deduce template parameters from constructor arguments.

As you can see, there are cases where the compiler cannot deduce the types for the template class—for example, when you use the default constructor (without arguments), as for `dblStuff`, or in the case of a constructor for which the compiler cannot deduce the type, as with `vcs`.

For some template classes, the compiler cannot or does not want to guess: for `ptr`, it could deduce `int` for `shared_ptr<int>` from the argument type `int*` of the constructor, but it does not do this automatically.

Deduction Guides

The compiler is smart enough to infer from a constructor argument of type `int` that the template parameter should also be `int`. However, if you look at `twoThree` in the

example, you will see that *iterators* and not *int* values are being passed. How does the compiler know not to build a `set<iterator>`?

This is done using *deduction guides*. In addition to the constructors, you write helper rules with `->`. I won't go into detail here. Essentially, for `set` and iterators, it will look like this:

```
template<class IT>
    set(IT, IT)
    -> set<typename std::iterator_traits<IT>::value_type>;
```

This means: “If the two constructor arguments are of the same type, check if `iterator_trait<IT>::value_type` exists for both, and if so, that is the type of the `set`.”

It is not always good to omit the arguments in the template class, even if the compiler could infer them. Sometimes it is better to name things explicitly. So use this new feature only if it makes your code *better*, not just *shorter*.

Moreover, only very modern compilers support this feature. Up to and including C++14, compilers could not use constructor arguments to deduce types for templates, so this was reserved for function templates. Therefore, many types in the standard library have defined `make_` helper functions to address this. Together with `auto`, you also get concise code.

```
auto tpl = std::make_tuple( 5, 'x' );
auto ptr = std::make_shared( 5 );
```

Listing 23.34 You have been able to use `make_` helper functions and `auto` for a long time.

Template Argument Deduction in This Book

In this book, I will prefer the new form because I believe that C++ is better, easier to understand, and easier to use with this innovation. If your compiler does not yet support *template argument deduction*, resort to using `auto` and functions or explicit specification.

23.5.4 Class Templates with Multiple Formal Data Types

As with function templates, you can essentially use any number of additional formal data types with class templates. You only need to list the formal data types in the definition of the class template between angle brackets, separated by commas, with the `typename` (or `class`) keyword. In the following example, you see a primitive version of `std::pair` as its own class template with multiple formal data types:

```
// https://godbolt.org/z/dMTnbT1v4
#include <iostream>
#include <concepts>    // copyable, C++20
template<std::copyable T, std::copyable U>
class MyPair {
    T data01_;
    U data02_;
public:
    MyPair(const T& t, const U& u) : data01_{t}, data02_{u} {}
    void print(std::ostream& os) const {
        os << data01_ << " : " << data02_ << std::endl;
    }
};
int main() {
    std::string month{"January"};
    int temp = -5;
    MyPair<std::string, int> temperature{month, temp};
    temperature.print(std::cout);
}
```

With the `MyPair<T, U>` class template, you can create a class that can manage two (almost) arbitrary types. In the example, I create an instance with the specific `std::string` and `int` types using `MyPair<std::string, int> temperature`. With `MyPair<std::string, int>`, the compiler instantiates the template and generates the class. The compiler then creates the `temperature` instance from this class. Instead of the C++20 `copyable` concept, you can also use `typename`.

Here's another tip, in case you do not want to instantiate a class template that contains multiple formal data types. An example would be `MyPair<double, double>`, where the compiler replaces `T` with `double` and `U` with `double`. The tip is, if you want to generate an instance of the class with `MyPair<double>`, you only need to adjust the template header as follows:

```
// https://godbolt.org/z/jbafEdoW
#include <iostream>
#include <concepts>    // copyable, C++20
template<std::copyable T, std::copyable U=T>
class MyPair {
    T data01_;
    U data02_;
public:
    MyPair(const T& t, const U& u) : data01_{t}, data02_{u} {}
    void print(std::ostream& os) const {
        os << data01_ << " : " << data02_ << std::endl;
    }
}
```

```
};
int main() {
    MyPair<double> numbers{11.11, 22.22};
    numbers.print(std::cout);
}
```

With typename **U=T**, you set U to the default value T if only one data type is written in the angle brackets and the second argument is missing, as is the case with `MyPair<double>`. Through this instantiation, the compiler replaces the formal data type T with `double`, and as the second type for the second formal data type U is not specified, U is also replaced with `double` due to `U=T`.

23.5.5 Class Templates with Nontype Parameters

A template parameter does not necessarily have to be a formal data type. Concrete data types (pointers and references are also allowed) can also be used. Such parameters are also referred to as *nontype parameters*. Here is an example of such a non-type parameter:

```
template<typename T, int val>
// ...
```

Here you see with `int val` such a nontype parameter, which can be used like a constant. However, this nontype parameter must be an integer type. In practice, this nontype parameter is often given a default value, which is used if no value is specified for this parameter during instantiation:

```
template<typename T, int val=10> // with default value
// ...
```

I want to create a simplified version of the fixed array `std::array` to demonstrate the nontype parameter. Here is the code, with an explanation following it:

```
// https://godbolt.org/z/E89bav3rE
#include <iostream>
template <typename T, size_t n=1>    // Nontype parameter with default value
class FixedArray {
    T data_[n] {0};                // Using nontype parameter
public:
    T& operator[](size_t index) { return data_[index]; }
    static constexpr size_t size() { return n; }
    void print(std::ostream &os) const {
        for(auto it : data_)
            os << it << ' ';
        os << '\n';
    }
};
```



```
    }  
};  
int main() {  
    FixedArray<int,10> vals {};           // n = 10  
    for(size_t idx=0; idx < vals.size(); ++idx) {  
        vals[idx] = idx+idx;  
    }  
    vals.print(std::cout);               // Output: 0 2 4 6 8 10 12 14 16 18  
    FixedArray<double> dvals;            // Default parameter for n  
    std::cout << dvals.size() << '\n'; // Output: 1  
}
```

For the `FixedArray` class template, besides the formal data type `T`, the nontype parameter `n` is defined, which here also gets the default value of 1. An array with 0 makes no sense and is not allowed anyway. With the `T data_[n]` member variable, this nontype parameter is then used to create a C-array with `n` elements. The class template is only instantiated here when it is used. For example, in `FixedArray<int,10>`, the formal data type is replaced by `int` and the nontype parameter `n` is replaced by 10. The compiler thus instantiates an `int` array with 10 elements. However, this fixed array offers more than a conventional C-array.

The static `size()` method returns the size of the array (whatever the nontype parameter is). Because `n` must be a compile-time constant, `size` can be static and `constexpr`. In addition, the method `print()` outputs all elements to a stream.

Finally, in `FixedArray<double>`, the formal data type `T` is replaced by `double`. Because no further specifications were made for the nontype parameter in the angle brackets and I set a default value of 1, a `double` array with one element is instantiated.

23.5.6 Class Templates with Defaults

As you have already seen, you can provide template parameters, like function parameters, with a default value. If the corresponding argument is missing when instantiating a template, the default value is used. The default values are usually specified in the definition of the template. However, default values in templates do not only apply to literal values; formal data types can also be preassigned with a default type, which is used if no type is specified during instantiation.

Default Values Also for Function Templates

Default values can also be used in function templates.

Returning once again to the class template `FixedArray`, you could, for example, use the following:

```
template <typename T=int, size_t n=10>
class FixedArray {
    // ...
};
```

Here I use the default `int` type for formal data type `T` and the default value 10 for non-type parameter `n`. These are always used when users of the class template do not provide any specifications during instantiation. The same rules apply here as with default values for functions. So if a parameter gets a default value, all other parameters to the right must also get a default value. Specifically, this means that if the formal parameter `T` has received the default value `int`, the nontype parameter next to it must also receive a default value.

From the `FixedArray` class template defined in this way with the two default values, a class can now be instantiated in various ways:

```
FixedArray<> vals;                // fixedArray<int, 10>
FixedArray<double> dvals;         // fixedArray<double, 10>
FixedArray<char, 8> bytes;        // fixedArray<char, 8>
FixedArray<100> whatever;        // ✗ Error!
```

From `FixedArray<>`, the compiler will generate a class with the default values that can hold an `int` array with 10 values as a property. With `FixedArray<double>`, however, the formal data type `T` from the class template is replaced by `double`, and the compiler generates a `double` array with 10 (default value) elements as a property. In the case of `FixedArray<char, 8>`, I use both a type for the formal data type and a value for the non-type parameter, resulting in the compiler generating a class that can store a `char` array with eight elements as a property.

Alias Template

An instantiation like with `FixedArray<100>` is not possible because if an argument is omitted during instantiation, all subsequent arguments must also be omitted. There is still a way to use such values with `using` in angle brackets:

```
template <size_t MAX>
using FixedIntArray = FixedArray<int, MAX>;
...
FixedIntArray<100> bigArray01;
```

Here, `FixedIntArray` is a so-called alias template. This allows you to generate a class for an `int` array, where you can specify the number, like 100 here, in the angle brackets. This class template can now be used with the `FixedIntArray` alias name. It is important to note that the `FixedArray<T, n>` class template is still being used. The call is simply made with an alias name to facilitate its use.

Standard Library

The standard library also makes your life easier with such alias names. Thanks to such using (or typedef), you don't have to use `std::string` as `std::basic_string<char>` because the following using was used (in a simplified and analogous manner):

```
using string = basic_string<char>;
```

23.5.7 Specializing Class Templates

Just like function templates, you can specialize class templates if a data type doesn't quite work with a class template or doesn't return a reasonable result. Unlike function templates, you can even partially specialize class templates (*partial specialization*), meaning you can specify only a few, but not all, of the formal parameters.

Partially Specializing Class Templates

To keep the topic from becoming too extensive, the `MyPair` class template will be used as an example again at this point. Specifically, I partially specialize the class template if one of the two formal data types contains `std::string` when called.

I partially specialize the following calls:

```
MyPair<int, std::string> data01{4, "Mars"};
MyPair<std::string, int> data02{"Toy Story", 1995};
```

And this is how partial specialization might look:

```
// https://godbolt.org/z/9rzsh5366
#include <string>
#include <tuple>
#include <concepts>           // copyable, C++20
template <std::copyable T, std::copyable U=T> // general case
class MyPair {
    T data01_;
    U data02_;
public:
    MyPair(const T& t, const U& u) : data01_{t}, data02_{u} {}
};
template <std::copyable T>           // partial specialization, T remains formal
class MyPair<T, std::string> {      // U is specialized to string
    std::tuple<T, std::string> data_;
public:
    MyPair(const T& t, const std::string& str) : data_{t, str} {}
};
```

```

template <std::copyable U>          // partial specialization, U remains formal
class MyPair<std::string, U> { // T is specialized to string
    std::tuple<std::string, U> data_;
public:
    MyPair(const std::string& str, const U& u) : data_{str, u} { }
};
int main() {
    // uses partial specialization
    MyPair<int, std::string> intString{1, "a"};
    MyPair<std::string, int> stringInt{"b", 2};

    MyPair<int, int> intInt{3, 4}; // uses general case
    MyPair intInt2{3, 4}; // also MyPair<int, int>
    MyPair<std::string, std::string> strStr{"c", "d"}; // ✗ ambiguous
}

```

First you see the original class template with two formal data types. Then you find the partial specialization that the compiler uses when the class template is used with `MyPair<..., string>` and a class is instantiated from it. Then follows the `MyPair<string, ...>` specialization. This demonstrates that the order in the partial specialization can also be reversed. Note that I stayed with `U` only for clarity, as it has nothing to do with the `U` from the general case. I could have also named the formal type parameter `X` or something entirely different. The general and the specialized template are independent of each other, just as there are, for example, two function definitions.

For simplicity in both cases, I have resorted to `std::tuple` as a data store to demonstrate that a specialized class template can be structured completely differently from the general template.

However, this example has a flaw if you want to use the class template with `MyPair<string, string>`. The compiler cannot instantiate now because it does not know whether to use `MyPair<T, string>` or `MyPair<string, T>`. If there were only one of the two partial specializations, the problem would be solved, and the one partial specialization would be used. If you called the other partial specialization, the compiler would generate an instance from `MyPair<T, U>`.

To avoid having to remove any of the partial specializations in this case, a complete specialization of `MyPair<string, string>` would also be an option.

What Cannot Be Partially Specialized?

In contrast to a class template, it is not possible to partially specialize a function template. A method of a class template also cannot be partially specialized; it must be fully specialized.

Fully Specialize Class Template

If none of the template arguments for class templates seem to fit, you can also create a full specialization of the class template. In the previous example, I mentioned a full specialization of `MyPair<std::string, std::string>`. You can implement a full specialization as follows:

```
// https://godbolt.org/z/baPeoTbM8
template<> // full specialization
class MyPair<std::string, std::string> {
    std::vector<std::string> data_;
public:
    MyPair(const std::string& t, const std::string& u) : data_{t, u} { }
};
int main() {
    // uses the full specialization:
    MyPair<std::string, std::string> strStr{"c", "d"};
}
```

To fully specialize a class template, the angle brackets in `template<>` remain empty, as with function templates. The data type or types for which the class template is now to be specialized must be written in the angle brackets after the class name, as seen here in `class MyPair<string, string>`. Through the full specialization of the class template `MyPair` by `MyPair<string, string>`, the specialization is now used by the compiler for generation after the `MyPair<string, string> strStr` call.

23.6 Templates with Variable Argument Count

You can create templates with any number of arguments for both functions and classes. These are called *variadic templates*. The syntax looks a bit unusual because the type list uses three dots `...` (an ellipsis). They serve here as a kind of operator.

Application Area

In practice, the implementation of such variadic templates is particularly interesting for those involved in creating libraries. However, if you want to create a template that can take a variable number of types, I would first recommend the sequential `std::tuple` data type (see [Chapter 28, Section 28.1](#)). Instead of writing a variadic template yourself, I recommend using a tuple as a parameter instead.

In this context, with `...` the parameters are packed into a parameter pack and also unpacked again. This depends on which side of the operator `...` is on. Reading the theory alone won't help much here, but a listing will demonstrate the matter to you.

```

// https://godbolt.org/z/eenKej9sj
#include <typeinfo>           // operator typeid
#include <typeindex>          // type_index
#include <map>
#include <string>
#include <iostream>
template <typename T>
void output(std::ostream& os, T val) { // output a type name
    // static: initialized the first time
    static const std::map<std::type_index, std::string> type_names {
        { std::type_index(typeid(const char*)), "const char*" },
        { std::type_index(typeid(char)), "char" },
        { std::type_index(typeid(int)), "int" },
        { std::type_index(typeid(double)), "double" },
        { std::type_index(typeid(bool)), "bool" },
    };
    const std::string name = type_names.at(std::type_index(typeid(T)));
    os << name << ": " << val << '\n';
}
// recursive variadic function template:
template<typename First, typename... Rest>
void output(std::ostream& os, First first, Rest... rest) {
    output(os, first);           // single call with the foremost element
    output(os, rest...);        // recursion with the rest of the elements
}
int main() {
    output(std::cout, 3.1415);   // normal template
    output(std::cout, "end", 2, 3.14, 'A', false); // recursive variadic function
}

```

Listing 23.35 A template with a variable number of arguments.

You should ignore the use of `type_names` with `type_index` and `typeid` here, it is only for informational output to indicate the type involved. You can find more about this in the explanation of [Listing 28.41](#) in [Chapter 28, Section 28.6](#).

In `main()`, I overloaded the `output()` function because it exists in two versions. But more on that shortly. First, consider the function calls to `output()`. The first function call invokes the nonvariadic template because I passed only one “arbitrary” argument besides `std::cout`. The second call uses more “arbitrary” parameters and therefore ends up with the variadic template.

In the nonvariadic case, there is nothing new—namely, no three dots ...

The variadic definition of `output` contains ... three times, each with a different meaning.

At the beginning, the operator appears in the list of formal template parameters:

```
template<typename First, typename ... Rest>
```

Here, it means that besides the `First` type parameter, there can be zero to any number of additional formal type parameters. They are all collectively referred to as `Rest` and can be addressed as such in the template definition.

Then follows ... in the function header:

```
void output(std::ostream &os, First first, Rest ... rest) {
```

Here ... stands between the formal type name and the name of the parameter `rest`. This stands for a whole list of parameters—all those after `first`. This is called a *parameter pack*.

In the function body, you can then “unpack” this parameter pack by using the `rest` variable followed by ...:

```
output(os, rest ...);
```

The compiler then turns this into a sequence of actual parameters at the moment of concrete use.

The important thing here is that `first` no longer appears. As you can see, it is a recursive call to `output`. Without `first`, the list of parameters is one shorter. The list is recursively shortened until there is only a single arbitrary parameter, because then the nonvariadic template fits. Its call ends the recursion.

Specifically, this means that the variadic call leads to a chain of the following calls:

```
// Call:
output(std::cout, "end", 2, 3.14, 'A', false);
// 1st Pass:
output(std::cout, "end");
output(std::cout, 2, 3.14, 'A', false);
// 2nd Pass
output(std::cout, 2);
output(std::cout, 3.14, 'A', false);
// 3rd Pass
output(std::cout, 3.14);
output(std::cout, 'A', false);
// 4th Pass
output(std::cout, 'A');
output(std::cout, false);
```

Loops Are Not Possible

Surely you have wondered why I did not use loops like `for` or `while` here. This is not possible because the parameter pack must be handled at compile time.

However, starting from C++17, there are several ways to avoid these sometimes hard-to-understand recursions. For example, the fold expression `... + CS` from [Listing 23.41](#) generate more compact code. In many cases, the new `if constexpr` can also help you.

Take a look at the program during execution:

```
double: 3.1415
const char*: ende
int: 2
double: 3.14
char: A
bool: 0
```

“operator<<” Available?

The reason our example works so smoothly here is that `operator<<` is defined for all the types I used. If that's not the case, you need to define `operator<<` for your type. Otherwise, the compiler would complain.

23.6.1 “sizeof...” Operator

There is indeed an operator with `sizeof...()` for parameter packs. With this, you can find the number of elements in the parameter pack at compile time. Here is the operator in action:

```
#include <iostream>
template <typename ... Args>
auto countArgs(Args ... args) {
    return (sizeof ... (args) );
}
int main() {
    std::cout << countArgs("one", 2, 3.14) << '\n'; // Output: 3
}
```

23.6.2 Convert Parameter Pack to Tuple

As I have already mentioned, `std::tuple` is much easier to handle than a parameter pack. Therefore, it is very useful that you can easily convert a parameter pack into a tuple using `make_tuple`:


```
// https://godbolt.org/z/8oYaWYnbx
#include <tuple>
#include <iostream>
template<typename ... Args>
auto conv2tuple(Args ... args) {
    return std::make_tuple(args...);
}
int main() {
    auto mytuple = conv2tuple("end", 2, 3.14, 'A', false);
    std::cout << std::get<2>(mytuple) << '\n'; // Output: 3.14
}
```

Simply pass `make_tuple` the parameter pack to be expanded—here, `args...`—and you get a corresponding tuple back.

23.7 Custom Literals

A *user-defined literal* is a custom *suffix* that can be appended to strings or sequences of digits. The suffix is a regular *identifier*. Users can define how a literal is interpreted by overriding a member of the new `operator""` family. The family members differ in the parameter types of the respective `operator""()`.

```
// https://godbolt.org/z/cjx49ef19
namespace my {
    Complex operator"" _i(const char*); // 0+ni
    Complex operator"" _j(long double); // n+0i
    Puzzle operator"" _puzzle(const char*, size_t);
}
using namespace my;
Complex imag4 = 4_i;           // operator"" _i(const char*)
Complex real3 = 3.331_j;      // operator"" _j(long double)

Puzzle p1 = "oXo"             // operator"" _puzzle(const char*, size_t)
           "XoX"_puzzle;
```

Listing 23.36 Custom literal operators.

The example assumes that you have defined the two `Complex` and `Puzzle` data types yourself. How is not part of the example and not relevant here. What the example shows is that you can write instances of these data types into the source code using literals like `4`, `3.331`, and `"oXoXoX"` and then get `Complex` and `Puzzle` instances instead of `int`, `double`, and `char[]`.

You must append a suffix to the usual literals. In the case of `Complex`, this is a `_i`, and in the case of `Puzzle`, it is a `_puzzle`. And how does `4` become a `Complex` or `"oXoXoX"` become a `Puzzle`? To do this, define operator `" "`s as free functions that take `4` or `"oXoXoX"` as arguments and return `Complex` or `Puzzle`.

That was the quick run-through—now comes the slow-motion version.

23.7.1 What Are Literals?

In the context that you can define them yourself, literals are

- sequences of digits mixed with a few special characters for commas or exponents, such as `7`, `3.14`, `0xff`, or `2.1e+4.8`; or
- strings enclosed in quotation marks, such as `"hello"` and `"octal\0123"`, optionally with
- a suffix, such as `L` or `d` for the exact data type, and/or
- for strings, the prefix `L` or one of the string prefixes for Unicode or raw string literals—`u`, `u8`, `U`, or `R`"(`.`

Then there are the few predefined literals that you cannot change or self-define. These are the *Boolean literals* `true` and `false` as well as the *pointer literal* `nullptr`.

Wouldn't it be nice if you could write [™] instead of the code sequence `\u2122` for a trademark symbol?

```
"Coca Cola{TRADE MARK SIGN}"_unicode
```

In the Unicode standard, each character also has a unique name.² In the International Components for Unicode (ICU) library, this is possible, but one does not want to bloat the compiler with the huge tables required for this.³ However, if users are already using ICU, they might want to take advantage of it for string literals.

In principle, programmers can now define the *suffixes* for literals themselves. And with the suffix, users specify which *free function* is used to interpret the literal in the source code.

23.7.2 Naming Rules

The standard restricts users in the choice of this identifier, so for portable code, you should adhere to the following rules for the identifier:

- The identifier must begin with an underscore `_`; all other literal suffixes are reserved for future language extensions.

² *Unicode 6.0 Character Code Charts*, <http://unicode.org/charts>

³ *International Components for Unicode*, <https://icu.unicode.org/>

- In particular, it should *not* begin with two underscores; the standard reserves this for the compiler and the standard library. Especially macros defined by the compiler or the library could conflict with this. Therefore, it is better to avoid this potential conflict.
- For the same reason, the identifier should not begin with an underscore followed by an uppercase letter—for example, not with `_A`.
- So the second character should be a lowercase letter or a digit.
- It is good practice to define new literal operators in their own namespace. This way, you can precisely control which operators are visible via `using namespace` directives. When you define the `_x` suffix, the `_x` identifier is not reserved: Only together with `operator""` does it become an identifier. Therefore, suffix `_x` and variable `_x` do not conflict with each other.

Tip

Operators for user-defined literals should be defined in their own namespace. Their names *must* start with an underscore and *should* continue with a lowercase letter.

23.7.3 Literal Processing Phases

To understand why there are different overloads of `operator""`, you need to know that the compiler processes literals in different phases. These are roughly the following:

- In the *token* splitting phase, the language elements of the source code are categorized into *keywords*, *numbers*, *symbols*, *strings*, and so on. Literals from this phase can be passed to `operator""` as *raw literals*. For numbers, these are then `const char*` (without `size_t` for the length) or the template variant; for string literals, they are the character string variants (with `size_t` for the length).
- Then numeric literals are converted into their numerical values. After this phase, they can be passed to the `operator""` variants with the numeric parameters and are then considered “cooked” (preprocessed—no longer “raw”).

23.7.4 Overloading Variants

All possibilities of using custom suffixes have in common that they are defined by `operator""`. Then follow the differences:

- **Type**
The *return type* is always the one that the literal should ultimately have. Both built-in types and user-defined types are possible.

- **Type operator `"" _x(const char*)`**

The `const char*` form without length parameters processes *raw* strings. A sequence of digits like `1234_x` is passed to operator `"" _x(const char*)` as a sequence of four characters *and* the terminating `'\0'` character. Note that this does *not* refer to string literals, even if the argument is `const char*`: *Raw* here means that the sequence of digits is not passed to the operator *preprocessed*. Because there are no quotation marks around the literal in the source code, you can only parse numeric literals this way. While you can write `234_x` or `9.87_x` in the source code, the compiler will complain about `hello_x` without quotation marks, as it has meanwhile classified this as an *identifier* token, not as a literal.

- **Type operator `"" _x(unsigned long long)`**

- **Type operator `"" _x(long double)`**

Preprocessed (cooked) numeric literals are processed with the parameters `unsigned long long` and `long double`. Thus, `4847_x` would result in a single call to operator `"" _x(unsigned long long)` with the decimal number 4847. The same call would also be made by `0x12ef_x`; the compiler cooks away the `0x` prefix and converts the number. Also, `10000.0_x` and `1+e4_x` lead to the same call to operator `"" _x(long double)`.

- **Type operator `"" _x(const CHAR*, size_t)`**

Where `CHAR` can stand for `char`, `wchar_t`, `char16_t`, or `char32_t`. These are preprocessed strings by the compiler and the only way to parse quoted literals. The operators receive two parameters, with the second specifying the length using `size_t`. The string does not need to be terminated by the `'\0'` character. *Preprocessed* here means that the compiler has already interpreted the prefixes before the string literals, such as `u8`, `R"(",` or `L`, and has already concatenated adjacent literals: `"mein" "text"_x` results in only one operator `"" _x(const char*,size_t)` call.

- **`template<char...> Type operator "" _x()`**

The *template* variant processes raw literals, sequences of digits without quotation marks. Unlike the `const char*` form, you can also use it for floating-point numbers. The literal `2.17_x` would lead to the call operator `"" _x<'2', '.', '1', '7'>()`—without the terminating `'\0'` character. Usually, you will define this operator recursively.

In practice, when the compiler decides which of these overloads to use, it looks like the following listing.

// <https://godbolt.org/z/Mevoshb3o>

```
namespace lits {
    long double operator"" _w(long double);
    string      operator"" _w(const char16_t*, size_t);
    unsigned    operator"" _w(const char*);
}
```

```
int main() {
    using namespace lits;
    1.2_w;      // operator"" _w(long double), with (1.2L)
    u"one"_w;   // operator"" _w(char16_t, size_t), with (u"one", 3)
    12_w;       // operator"" _w(const char*), with "12"
    "two"_w;    // ✗ operator"" _w(const char*, size_t) not defined
}
```

Listing 23.37 Which literal leads to which operator call?

For the literal `"two"_w`, the compiler looks for `operator"" _w(const char*, size_t)`. Because we have not defined that, the compiler outputs an error message there.

23.7.5 User-Defined Literal Using Template

The template variant is also easy to implement if you only need to handle a well-defined set of literals.

```
// https://godbolt.org/z/5rdzn8YP4
namespace lits {
    // general template
    template<char...> int operator"" _bin2();
    // specializations
    template<> int operator"" _bin2<'0','0'>() { return 0; }
    template<> int operator"" _bin2<'0','1'>() { return 1; }
    template<> int operator"" _bin2<'1','0'>() { return 2; }
    template<> int operator"" _bin2<'1','1'>() { return 3; }
}
int main() {
    using namespace lits;
    int one   = 01_bin2;
    int three = 11_bin2;
}
```

Listing 23.38 A template for literals of exact length two.

First, the general template is declared, but without implementing (“defining”) it. Here we definitely need a variadic template with the template argument `char...` for `operator""`. The definitions then follow in a list of template specializations for exactly the literals you want to use.

Here, `0_bin2` and `11111_bin2` would naturally cause an error because we have not defined any specializations for them. Moreover, this type of enumeration is tedious and error-prone—all in all, not recommended. But it leads to the concept of defining the template *recursively*.

```

// https://godbolt.org/z/sbxojqjqs
namespace lits {
    // Template helper function for one argument
    template<char C> int bin(); // general case
    template<>      int bin<'1'>() { return 1; } // Specialization
    template<>      int bin<'0'>() { return 0; } // Specialization
    // Template helper function for two or more arguments
    template<char C, char D, char... ES>
    int bin() {
        return bin<C>() << (sizeof...(ES)+1) | bin<D,ES...>(); // Bit-Shift, Bit-Or
    }
    // actual operator"" _bin
    template<char...CS> int operator"" _bin()
    { return bin<CS...>(); };
}

int main() {
    using namespace lits;
    int one = 1_bin;
    int eight = 1000_bin;
    int nine = 1001_bin;
    int ten = 1010_bin;
    int eleven = 1011_bin;
    int one_hundred_twenty_eight = 10000000_bin;
}

```

Listing 23.39 Recursive definition for arbitrary literal lengths.

The actual work of operator"" _bin has been offloaded to the normal (recursive template) bin helper function. This is simply called with bin<CS...>(), and the result is passed through.

The actual conversion is then carried out there. Here, the two specializations on <'0'> and <'1'> serve as *anchors*, where the recursion ends. In the general helper function, a character is always converted with bin<C> and shifted to the right by as many bits as there are template arguments left using <<. This is determined with sizeof...(ES). The recursive call bin<D,ES...> then converts the rest of the 1/0 string without the leading character C. Both parts are combined with bitwise OR |.

This recursion is already executed by the compiler at compile time. This gives the best chance that in the translated program, only the calculated constant remains for each _bin literal and no calculations need to be performed at runtime.

Avoiding Recursion with “constexpr-if”

With C++17, you can simplify code with multiple specialized function templates into a single general function template. You do this with `if constexpr(...)`: The condition specified in parentheses must be evaluable at compile time. You can then combine the first three helper definitions from the previous example as in the following listing.

```
// https://godbolt.org/z/KvsGh4f1v
namespace lits {
    // Template helper function for one argument
    template<char C> int bin() { // general case
        if constexpr (C=='0') return 0;
        else if constexpr (C=='1') return 1;
    }
    // Template helper function for two or more arguments
    // ...
}
```

Listing 23.40 The new `if constexpr` saves function specializations.

The `C==0` and `C==1` expressions are evaluable at translation time. `bin<'0'>` and `bin<'1'>` can therefore be computed by the compiler.

The special thing about this `constexpr-if` is that it behaves as if you had written different template specializations: code from then branches that you never need in the entire program is not instantiated by the compiler. It does not appear in the translated program and does not affect its size or runtime.

With Fold Expressions to One-Liners

Admittedly, the recursive template definition from [Listing 23.39](#) is not easy to digest. Such constructs with helper templates and partial specializations have been a thing of the past since C++17. I just want to give a small insight into *fold expressions*, which will simplify a lot in the implementation of variadic templates.

The implementation in [Listing 23.41](#) only converts the literal into a string that looks exactly like the literal.

```
// https://godbolt.org/z/cPoEjqzM7
#include <iostream>
#include <string>
namespace lits {
    // operator"" _sx
    template<char...CS> std::string operator"" _sx() {
        return (std::string{} + ... + CS); // a fold expression
    };
}
```

```
int main() {
    using namespace lits;
    std::cout << 10000000_sx << '\n';    // Output: 10000000
    std::cout << 10'000'000_sx << '\n';    // Output: 10'000'000
    std::cout << 0x00af_sx << '\n';        // Output: 0x00af
    std::cout << 0x0'c'0'a'f_sx << '\n';    // Output: 0x0'c'0'a'f
    std::cout << 007_sx << '\n';           // Output: 007
    std::cout << 0b01_sx << '\n';          // Output: 0b01
}
```

Listing 23.41 Fold expressions make variadic templates simpler.

In the listing, *(init+ ... +pack)* is the *fold expression*. What happens here is that `template<'1','0','0','0'>` is folded at compile time into the expression `string{}+'1'+'0'+'0'+'0'`. This is a powerful feature that makes some templates shorter.

The *init* part of the fold expression can be any expression you specify. The other two parts, the points `...` and *pack*, are derived from the template arguments during the call or template instantiation. The *init* part is optional. It can be specified at the beginning or the end. Instead of `+`, all binary operators are allowed.

23.7.6 Raw or Cooked

The difference between raw and preprocessed literals is that the former always get the `const char*` variants as arguments (without the length) and the latter are already interpreted as unsigned long long or long double. Both can process literals like 1234 and 23.45. The `const char*` operator, which also gets the length as a parameter, is intended for processing string literals like "abcd".

So, raw or preprocessed makes a difference for numeric literals.⁴

```
// https://godbolt.org/z/Yd4s5ser5j
#include <complex>
// raw form
int operator"" _dual(const char*);
int answer = 101010_dual;    // decimal 42
// preprocessed form
std::complex<long double> operator"" _i(long double d) {
    return std::complex<long double>(0, d);
}
auto val = 3.14_i; // val = std::complex<long double>(0, 3.14)
```

Listing 23.42 Whether preprocessed or raw is better depends on the application.

⁴ What new capabilities do user-defined literals add to C++?, users Motti & hichris123, <http://stackoverflow.com/questions/237804/>, Stack Overflow, 2008 & 2013, [2024-08-12]

You can decide whether you prefer to receive a numeric literal as a preprocessed value or as a sequence of characters. For converting a sequence of 0/1, a character sequence is probably more practical. Preprocessed, the compiler would have passed the decimal number “one hundred eleven” to `operator""_dual`, and we would have had to interpret it in binary with effort. We already saw the template alternative with `operator""_bin`.

Conversely, it is very convenient for `operator""_i` to simply use the constructor of `complex` that takes a `double` as an argument. If we had received the literal `raw` as `const char*`, we would have had to convert it to a number ourselves.

23.7.7 Automatically Merged

Merging adjacent literals always carries some risks: Did you just forget the separator, or is it the user's intention to keep the source code nicely formatted? In principle, nothing has changed here because the compiler has always done this task for us.

Only the task has become much more complex due to the many suffixes and prefixes. The rule for strings with user-defined suffixes states something similar to prefixes:

- If all strings have the same suffix, the result is unambiguous.
- If adjacent strings have different suffixes, it is an error.
- If only one of the concatenated strings has a suffix, the entire string is interpreted as if it had that suffix. Which of the strings has the suffix is irrelevant.

The standard provides some examples to illustrate this:

- `L"A" "B" "C"_x`; becomes `L"ABC"_x` when concatenated.
- `"P" _x "Q" "R" _y`; results in an error because two different suffixes are used.

23.7.8 Unicode Literals

The variants with string length are invoked for literals with the different *encoding prefixes*. That means, depending on `CHAR`, you write this variant:

Type `operator""_x(const CHAR*, size_t)`

Which `CHAR` you have to use depends on the *encoding prefix* of the of your string, as listed in [Table 23.2](#).

Literals in normal and UTF-8 encoding both call

`operator""(const char*, size_t)`

to be interpreted.

Prefix	Example Literal	CHAR
	"Nodding dog"	char
u8	u8"B\x3c\xbcgeleisen"	char
L	L"As usual"	wchar_t
u	"B\u00fcgeleisen"	char16_t
U	"B\u0000üfcgeleisen"	char32_t

Table 23.2 Encoding prefixes and the corresponding overloads.

Pitfall

When defining a custom operator `""_suffix()`, it is important to ensure that a space follows the `""`. If this is missing, the parser will recognize it as a single token and stop with a syntax error. Also, no character is allowed between `""` and `""`—not even spaces.

A space is allowed between the operator and `""`, but it is not required. The mnemonic here is that `operator""` becomes a single identifier—as with all operators—and therefore no space is allowed within it.

PART IV

The Standard Library

The standard library is an integral part of the C++ language. It contains indispensable and useful features. Always use the standard library, rather than developing yourself or turning to third-party libraries.

In this part, you'll get to know the various parts of the standard library in more detail. I dedicate quite a lot of space to containers, iterators, and algorithms. This is followed by a chapter on streams, which also work with iterators. You will then learn how to program threads in C++. Finally, we'll discuss various smaller library elements.

With these chapters, you'll gain deep insight into the standard library, so you'll need to refer to the reference less often.

Chapter 24

Containers

Chapter Telegram

- **Container**
A concept of the standard library, according to which multiple types operate. They hold elements of the same type.
- **Sequence containers array, vector, deque, list, and forward_list**
Order of elements determined by the user.
- **Associative containers map and set, each also multi and unordered**
For storing a value based on a key.
- **Adapter**
A group of wrapper classes around containers that offer their own interface.
- **Spanning views span and string_view since C++20, and mdspan since C++23**
Configurable views into a container.
- **<algorithm>**
Standard library header with extended operations on containers.
- **Iterator**
A concept for referencing elements in containers.
- **Allocator**
The ability to intervene in the memory management for the elements of a container.
- **Search tree**
An associative, sorting data structure with guaranteed fast access to the elements.
- **Hash table**
An associative, nonsorting data structure for very fast access to elements. The performance is not guaranteed and depends on many factors.
- **bitset**
Data structure for storing bits; not really a container.
- **valarray**
Special data structure for matrix computation; not really a container.

Containers form the core of the standard library. The part that dealt with containers, iterators, and algorithms was originally called the Standard Template Library, or the STL for short. The research department of Hewlett-Packard (HP) introduced the STL

into the standard around 1993, thus providing the first major building block of a standard library that truly looked like C++. Because a large part of the STL consisted of containers and their surrounding components, the terms *STL* and *standard library* were used synonymously for a long time. It was only with C++11 that the non-STL part of the standard library grew to the extent that it was slowly recognized that what was originally the STL is now just a part of the standard library. And even in that part, a lot has changed since its origins.

24.1 Basics

However, the *containers* are still around and much of the standard library revolves around them. Containers are a collection of recurring data structures that offer standardized operations and functions independent of their own data types. The property of being independent of the data type is called *abstract*. The fact that they recur makes them extremely useful—with an emphasis on *extremely*.

24.1.1 Recurring

In the 1970s, it was estimated that 80% of the world's computer time was spent sorting data. That may sound like a lot, but depending on how you count, it might even be an underestimate—and it is still true today, especially in modern times. In a broader sense, *searching* can also be seen as a subproblem of *sorting*. When you consider the effort Google alone must put into its search engine to enable us to find web pages and offer “relevant advertising,” the estimate might indeed be on the low side.

And if so much effort is put into searching and sorting, then it is only likely that you will encounter this problem at one point or another in your career. Therefore, here is my request: do not reinvent the wheel if you do not absolutely have to. The probability is high that someone has already done this work for you, and the probability is still high that a solution is available to you in the standard library.

Not everything revolves around sorting, not everything around searching, but a surprising amount does. Do you need to manage memory for 1,000 elements? Do you want to implement a buffer for the database? Do you want to eliminate all duplicates? Yes, you want to search and sort. But what does that have to do with the containers of the standard library? Quite simply, someone before you had the same tasks to tackle and found a solution for them. Then another, then another, and then the following turned out to be true:

- Many elements are best placed sequentially in a loosely coupled list or tightly packed in a row, in an *array*. Mathematicians think of this as a *vector*.
- A buffer should deliver things that come in first, behaving like a *queue*.

- Duplicates are best found when the elements are kept sorted, either during insertion or once at the end.
- You always put something in a *container*, but depending on the task, you use different types of containers. But what exactly is a container? It has something inside it. Anything more? Can you always put something into a container? Always take something out? Where can you put it in? Front or back? Or even in the middle? And if I can take something out—to where? In general, how does finding work?

First, you can identify groups of different containers with different properties:

- Tightly packed or loosely packed?
- Always sorted or not? And how can you search them?
- Adding and removing happens at which positions?

And then you realize that the whole thing can be done with a handful of different containers—if they are designed flexibly enough.

24.1.2 Abstract

This means primarily that it must not matter to you *what* is put into the containers. A basket must always fulfill its task, whether it contains tomatoes or yogurt cups. Or in C++ terms: it is irrelevant whether the vector is filled with `int` or `class Student`. The class or built-in type on which the container operates as an abstract data type is specified in C++ as a template parameter in angle brackets:

```
std::vector<int> many_int_values {};  
std::vector<Student> many_student_values {};
```

As a reminder, the type of the container includes all template arguments. The data type `vector<int>` is different from `vector<Student>`, and `array<int,4>` denotes a different data type than `array<int,7>`. This means, for example, that you can write four different overloads of a function that take these types as parameters:

```
void show(vector<int> arg);  
void show(vector<Student> arg);  
void show(array<int,4> arg);  
void show(array<int,7> arg);
```

You will see that the standard library does not work with overloads of this kind. Instead, you use `span`, `input_range`, or `view` to pass the elements to be output:

```
void show(span<const int> arg);           // for vector<int> and array<int, ...>  
void show(span<const Student> arg);       // for vector/array<Student>  
void show(ranges::input_range auto&& arg); // for all Views and Containers
```


You can also pass a pair of iterators that represent the beginning and end of the elements to be output. However, with the availability of `span` and the various types of ranges, this has become uncommon.

With iterators, it looks as follows, with an abbreviated function template using `auto` and the `input_iterator` concept—both available from C++20:

```
void show(input_iterator auto begin, input_iterator auto end);
```

Without C++20 features, you make the template explicit and without concepts:

```
template<typename IT>  
void show(IT begin, IT end);
```

Instead of the universal reference `&&`, you can also take a value parameter. However, it will be copied when called, which you probably don't want for a real container with potentially many elements. Then you should restrict the overload to the `view` concept:

```
void show(ranges::view auto arg);
```

This prevents the call to `show(vector{1,2,3})`. If you want to enforce this, convert the container with `all` to `show(views::all(vector{1,2,3}))` explicitly into a view.

Ranges as parameters are a complex topic. You can find more details in [Chapter 25, Section 25.6](#), under the subheading Ranges as Parameters (and More).

24.1.3 Operations

Each of the containers is associated with the operations it must be able to perform. Over time, more or less established names have emerged for this. Let's take as an example one of the simplest abstract data types, the *stack*.

A stack can (theoretically) hold an unlimited number of elements. These are added with `push()`. At the end where you add, you can remove the most recently added element with `pop()`. This can be done until the stack is empty again, which is checked with `empty()`.

As minimum operations, this is sufficient for a stack. You can request additional “luxury operations,” but the minimum set is provided with this.

Do you want to know how many elements a given stack contains—that is, you want to know `size()`? Then copy all elements with `pop()` and `push()` to another stack until the original is `empty()`, count them, and when you are done, stack everything back. This is certainly not a fast method, but feasible in principle. Whether you want to include `size()` as a mandatory operation in your stack or not and whether `size()` needs to be fast or not is your decision and depends on the task.

The standard library specifies which containers exist and what interfaces they have. It defines a list of abstract data types and specifies which (minimum) operations can be

performed on them and what constraints apply. In the case of `stack`, for example, the standard library specifies that there is a `size()` method and that this method must not internally touch all elements but must return immediately.

All containers strive to offer all operations under the same name. If a method does not exist for a container, it means that this method would conceptually consume many resources for this container—for example, `size()` in `forward_list`. But even if it exists, it is worth looking into the documentation to find out if the method might be expensive, such as `insert()` in `vector`.

You will find the methods and operations for all containers in this book. I place importance on clarity and practical advice. What I cannot and do not want to provide is a dry list of all methods and free functions with all their overloads for each individual container. For one, this list would be hard to read—you can find it in the standardization document—and for another, it would simply be too long and confusing. The list of constructor overloads for `vector` alone would fill a page in the table layout of this book. Therefore, I have decided to offer you two other types of presentations instead: a prose description filled with examples for each container and, in addition, a compact overview of almost all method names in a concise table, also for comparing the containers with each other.

24.1.4 Complexity

What does *expensive* mean? The specification of the containers determines how many resources their operations are allowed to consume, such as memory and time. This resource consumption is always given in relation to the number of elements n in the container. Not specifically in bits, bytes, or seconds and days, but in a function that grows at different rates depending on this n . The function is a measure of *complexity* and is indicated with $O(\dots)$.

This *O-notation* is an indication of how many resources an algorithm typically consumes. The mathematical function given there describes a curve: the faster it grows, the greater the resource consumption with increasing input. Typical complexity indications in order of their maliciousness are as follows:

- **$O(1)$**
Constant resource consumption. No matter how large n is, the operation always takes the same amount of time. Accessing a `vector`, for example, is constant.
- **$O(\log n)$**
Logarithmic resource consumption. n has to become very large before many resources are consumed. Searching in a `set` is logarithmic.
- **$O(n)$**
Linear resource consumption is usually harmless unless you run the code over and over again. Searching in an unsorted `vector` is linear.

- **$O(n \log n)$**
Many sorting algorithms fall into this complexity class and are still considered harmless for datasets of normal size.
- **$O(n^2)$**
Quadratic resource consumption is already problematic. Two normally nested loops quickly fall into this category.
- **$O(2^n)$**
You should avoid exponential resource consumption like the plague. Finding the shortest path through n cities is such a problem.

Therefore, you should always strive to perform an operation with the smallest possible complexity. This can make the difference in the success of your program.

However, things get a bit more complicated with *amortized costs*. For example, `map::erase()` states that the amortized cost is $O(1)$. This means that if you perform the operation frequently, it averages out to $O(1)$, but a single operation might take longer. So, if you have a `map` with n elements and you use `erase()` to delete all n elements one by one, it will have taken $O(n)$ time in total—amortized.

One other notable and important operation with low amortized costs is `vector::push_back()`. If you sequentially insert $O(n)$ elements into a `vector` using `push_back()`, it takes a total of $O(n)$ time. Each individual `push_back()` takes between $O(1)$ and $O(n)$ time, the latter because `vector` occasionally copies all existing elements when it grows. However, this happens in such a clever way that on average—amortized—it results in $O(1)$ per `push_back()`. Nevertheless, if you know in advance how many elements you will insert with `push_back()`, you can still save time and space by calling `reserve()`.

Input Size	Quicksort		Bubblesort	
	Steps	Approx. Time	Steps	Approx. Time
n	$O(n \log n)$		$O(n^2)$	
10	33	0 sec.	100	0 sec.
100	664	0 sec.	10000	0 sec.
1000	9965	0 sec.	1000000	1 sec.
10000	132877	0 sec.	100000000	100 sec.
100000	1660964	1 sec.	10000000000	3 hrs.
1000000	19931568	19 sec.	1000000000000	12 days

Table 24.1 The runtime of Quicksort and Bubblesort for different input sizes, assuming the computer performs about 1,000,000 operations per second.

If we go back to sorting at the beginning, you can choose between several sorting methods, such as *Quicksort* and *Bubblesort*. Quicksort sorts n elements in $O(n \log n)$ steps, while Bubblesort takes $O(n^2)$. In general, you should never choose *Bubblesort*; if you do, your program might feel like it will never finish. If you can't imagine what $n \log n$ and n^2 mean, see [Table 24.1](#) for specific values.

Regarding the containers and their operations, this means that you can look up in the cheat sheet in [Appendix A](#) which complexity class the operation falls into. The implementation thus guarantees not only that the operation is performed correctly but also that it is done with a guaranteed limited resource consumption.

24.1.5 Containers and Their Iterators

One of the basic principles of containers is that all operations refer either to a single element or to a range. Individual elements are *always* passed by value (call-by-value), in exceptional cases perhaps as a constant reference. Handles into a container are extremely rarely references or even pointers, but usually *iterators*. Such an iterator is just a more general idea of a pointer, as the operations allowed on it are mainly as follows:

- Dereferencing with `it->...` and `(*it)`
- Comparison with `it1 == it2`
- Increment with `++it` (or `it++`)

Some iterators can also be decremented (`--it`) and some can be moved arbitrarily (`it+10` and `it-22`), but not all.

Each container type has public inner type aliases for iterators that refer to its elements. You get them with `container::iterator` and `container::const_iterator`—for example, with `vector<int>::iterator` and `vector<int>::const_iterator`. Thanks to `auto`, you rarely have to write it out.

The main use of iterators in the standard library is that they define the *area*: A pair of iterators pointing to the same container defines an area of elements in that container. Thus, there are elements *within* the area defined by the iterators and elements *outside*. I chose the word *area* here because `range` is now used for something specific, which I will explain very soon.

Every container has the methods `begin()` and `end()`. Both return an iterator of the respective container. Between `begin()` and `end()` lie exactly and precisely *all* elements of the container.

A method or function that takes a pair of iterators as an argument operates on the elements of this range.

For such a *pair* of iterators, since C++20, there are much better *ranges* from the `<ranges>` header or the *extensions* span from the `<span>` header. Whenever you work with a pair of iterators, you should use a span or a range—in the latter case, usually together with `auto` and the concepts from `<ranges>`—like, for example, `input_range`. Most often, it is the constructors and insertion methods that you can call with a pair of iterators or a range. For some insertion methods, since C++23, there is also a range variant. You will learn more about ranges in the next subsection.

```
// https://godbolt.org/z/qGrYPrqYn
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector nums{ 1,2,3 };
    vector four{ 4,5,6 };
    vector seven{ 7,8,9 };
    nums.insert(nums.begin(), four.begin(), four.end()); // pair of iterators
    cout << nums.size() << "\n";                       // Output: 9
    nums.insert_range(nums.begin(), seven);              // C++23: range
}
```

Listing 24.1 For parts of containers, use a pair of iterators, or if you use C++23, ranges.

Return values from container methods like `equal_range()` are still pairs of iterators because in the `<ranges>` and `<algorithms>` headers, there are range variants as free functions.

The `erase()` method of the `vector` class takes a pair of iterators as an argument and deletes the range defined by them. Therefore, if you apply `erase()` to `begin()` and `end()`, it should not be surprising that the container is empty afterward.

```
// https://godbolt.org/z/M1d9jEP74
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector numbers{ 1,2,3,4,5 };
    numbers.erase(numbers.begin(), numbers.end());
    cout << numbers.size() << "\n"; // Output: 0
}
```

Listing 24.2 A pair of iterators defines a range of elements.

The `begin()` and `end()` methods of containers are also indirectly used by the *range-based for loop*:

```
// https://godbolt.org/z/Yv3vcxPv5
#include <vector>
#include <iostream>
int main() {
    std::vector<int> numbers{ 1,2,3,4,5 };
    for(auto val : numbers) {
        std::cout << val << ' ';
    }
    std::cout << '\n';
}
```

Listing 24.3 Indirectly uses `begin()` and `end()` of the container.

This `for` is shorthand for the following:

```
// https://godbolt.org/z/babK485oT
#include <vector>
#include <iostream>
int main() {
    std::vector<int> numbers{ 1,2,3,4,5 };
    for(auto it = begin(numbers); it != end(numbers); ++it) {
        auto val = *it;
        std::cout << val << ' ';
    }
    std::cout << '\n';
}
```

And the overloads of the `begin(...)` and `end(...)` global functions for containers do “nothing” other than return `container.begin()` and `container.end()`.

An iterator that is not defining one boundary of an area often serves as an indirection to the container content. The return value of the `find()` method, which some containers have, is, for example, an iterator. If this is identical to `end()`, then the searched-for value was not found. Otherwise, you get the searched value by dereferencing with `*`:

```
// https://godbolt.org/z/nj14e6Wq4
#include <set>
#include <iostream>
int main() {
    std::set<int> numbers{ 10, 20, 90 };
    auto no = numbers.find(30);
    if(no == numbers.end()) { std::cout << "not there.\n"; }
```

```
auto yes = numbers.find(20);
if(yes != numbers.end()) { std::cout << *yes << '\n'; }
}
```

24.1.6 Ranges Simplify Iterators

To understand containers, you can't avoid iterators. They are not difficult; they abstract pointers from the basic idea. However, iterators can be associated with additional functionalities and properties: how they are incremented, whether they allow random access, and so on. This provides a clear interface to algorithms, which we will discuss shortly. But they also have their weaknesses. For nearly 30 years, we have lived with them, and improvements have been worked on for a long time. Since C++20, one such improvement has arrived: *ranges*.

Ranges are a new way to define ranges of elements. They are not limited to containers. However, when it comes to containers, they bundle the two iterators that are always needed together. This can be problematic; for example, if you accidentally mix `begin()` and `end()` from two different containers, the compiler will not notice. Ranges are passed as a single parameter. This is not only a simplification but also a unification, as ranges combine the following types of ranges into one concept:

- **first and (exclusive) last**

A range between two iterators, as containers provide with `begin()` and `end()`.

- **first and size**

A range *from* an iterator with a specific length, typical for C-arrays.

- **first and a predicate**

Something like `<iostreams>`, such as `istream_iterator`, which reads elements from a stream until a certain predicate is met.

- **first and ...**

Generators like `iota()` from `<numeric>`, which produces an endless sequence of elements.

A range can represent all of this. And in some cases, we don't need to know which of these variants underlies a given range. When you browse the documentation, you will encounter the term *sentinel*. This is the generalization of the `end()` iterator and is part of the range definition. The technical difference between the `end()` of containers and the sentinel or ranges is that the sentinel does not have to be of the same type as the `begin()` of the range.

A typical example is `'\0'` at the end of C strings or the EOF in files. If you want to classically map an iterator `end()` for these cases, you first have to run to the end of the container. This is not only inefficient but also impractical for EOF in files. In the concept of the sentinel, this now abstractly reads as “until `'\0'`” or “until EOF.”

Spans Are Different from Ranges

For contiguous containers like `string` and `vector`, simpler variants of views into containers are possible: `string_view`, the span that has existed since C++20, and the `mdspan` added with C++23. These should not be confused with ranges and their views. However, a span is also a `borrowed_range` and therefore compatible in many places.

Combining Views and Pipelines with Ranges

But that's not all. With ranges, *views* were introduced, and with views, pipelines. Are you coming from Java? Do you like streams? Functional programming? Composition? Then you'll love views. Views are ranges that do not store elements themselves but access other ranges. So they are a kind of “perspective” on other ranges. And pipelines are a way to combine views. So they are something like a connection-cable for views.

With the pipe operator `|`, you create and combine views. We will get more specific about algorithms in a moment, but I want to briefly show you what a pipeline of views looks like in code. The next listing shows how this previously looked with algorithms.

```
// https://godbolt.org/z/arfee3TTP
using namespace std;
vector in { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector<int> tmp{};
vector<int> out{};
copy_if(in.begin(), in.end(), back_inserter(tmp), [](int i) { return i%3 == 0; });
transform(tmp.begin(), tmp.end(), back_inserter(out), [](int i) {return i*i; });
```

Listing 24.4 One algorithm after another with iterators.

And in the next one, you can see that it is significantly more elegant with a pipeline of views.

```
// https://godbolt.org/z/TG1qd31Mq
using namespace std; namespace views = std::ranges::views;
vector in { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto out = in
    | views::filter([](int i) { return i%3 == 0; })
    | views::transform([](int i) {return i*i; });
```

Listing 24.5 A pipeline of views instead of algorithms.

With the first `|` I create a view into the container `in`. The adapter `filter` creates another view that only allows elements that satisfy the predicate. None of the views store

elements. They are just views on the elements of the container. The second `|` combines this view with the `transform` range adapter—a computation on the individual elements.

Even the `out` variable is initially a view, and you should output or further process it as it does not contain real elements. This is important because so far the preceding code snippet calculates *nothing*. Views behave in a *lazy* manner: only when you query the elements of `out` are the computations performed.

C++23 and Ranges Version 3

With C++23, the standard library has received a massive extension of ranges. You get many more ranges and views to combine. Ultimately, starting with C++23, working with ranges becomes truly enjoyable. I will point out elements from C++23 in the book.

Without C++23, you either use a loop or iterators to materialize the elements of a view—for example, to store them in a container:

```
vector result(out.begin(), out.end());
```

As you can see, in C++20, you even have to resort to the iterators of the view to fill the container. Starting with C++23, you can also use `ranges::to` within the pipeline for this.

```
// https://godbolt.org/z/EdYfzzdEM
// ... as before ...
auto out = in
    | views::filter([](const int i) { return i%3 == 0; })
    | views::transform([](const int i) {return i*i; })
    | std::ranges::to<vector>;
```

Listing 24.6 With C++23, you can use `ranges::to`.

However, as of November 2023, neither g++, Clang, nor MSVC is able to compile it reliably. You can use the `__cpp_lib_ranges_to_container` macro to check if the compiler supports `ranges::to`.

Using the `from_range` marker as a constructor parameter, it also works directly with the range in C++23.

```
// https://godbolt.org/z/veovKc8Kj
// ... as before ...
auto out = in
    | views::filter([](const int i) { return i%3 == 0; })
    | views::transform([](const int i) {return i*i; });
vector<int> result(from_range, out);
```

Listing 24.7 The `from_range` marker also exists only since C++23.

24.1.7 Ranges, Views, Concepts, Adapters, Generators, and Algorithms

The ranges library is located in the `<ranges>` header, contains the `std::ranges` namespace, and is divided into several areas:

- **Ranges**

A *range* is essentially anything that can be iterated over, has a start, and potentially an end, called a *sentinel*. A typical range is based on a data area—for example, a container—and you can also modify the elements through the range.

- **Views**

A *view* is a range and is defined by operations it wants to perform on the elements, such as filtering or transforming the elements. Views are *lazy*, performing operations only when necessary.

- **Concepts**

The ranges library defines, for example, the `input_range` concept, which you can use in function parameters.

- **Converter**

If you use `ranges::to` (C++23), a view is converted into a container.

- **Factories**

Factories are views that do not rely on underlying data but generate values—for example, `iota_view`, `empty_view`, or `cartesian_product_view` (C++23).

- **Adapter**

An adapter takes a range and produces a view of a specific type from it. For example, the `reverse` adapter produces a `reverse_view`. Typically, you use adapters with the pipe operator `|`.

- **Generators**

With `ranges::generator`, you can use coroutines.

- **Algorithms**

Most algorithms that previously worked with containers are now also defined for ranges. They always work *eagerly* (i.e., not *lazily*). You can find them in the `<algorithm>` header in the `std::ranges` namespace. They are called *constrained algorithms* there.

You have only received a small insight into the ranges library here. I have intentionally placed it *before* the brief insight into the algorithms library so that you will already know how to combine containers and ranges there.

I go into the details of the ranges library in [Chapter 25](#).

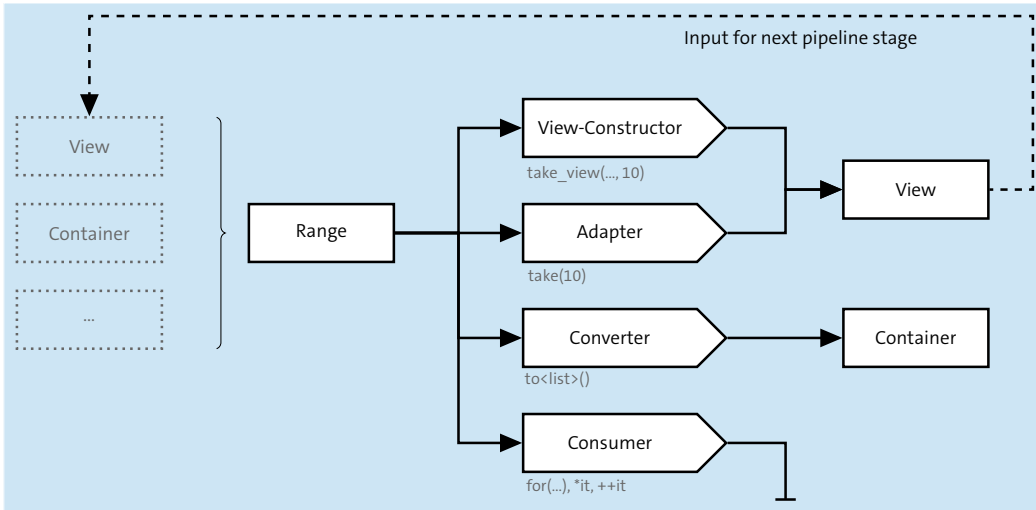


Figure 24.1 Containers and views are ranges, and adapters generate views from them that can be further processed.

24.1.8 Algorithms

The original STL also includes the algorithms that operate on the containers. These are a collection of free functions that are there to support the methods of the containers. However, as parameters, they do not receive a container, but always *pairs of iterators*. Since C++20, many algorithms are also defined in `std::ranges` and operate on ranges. This achieves two effects:

- Algorithms always work on *subranges* of containers.
- Algorithms work almost independently of the container type on iterators of all containers.

The latter is only true in principle. There are often constraining conditions. For example, you can apply `sort(begin, end)` to an iterator pair of a vector, but `sort()` does not work on `list` iterators. They do not allow random access. To distinguish, containers and iterators are divided into groups. If you want to work with a free function for iterators, check the following section using [Table 24.2](#) to see if the category of the iterator matches.

24.2 Iterator Basics

Iterators fall into different categories regarding the operations they support. This is a simple linear hierarchy: the higher the category in this hierarchy, the more operations that iterators of that category support.

The exception is the *output iterators*, which require additional properties but do not necessarily bring any of the other categories with them.

Iterator Category	Provider
Contiguous	vector, string, array
Random access	deque
Bidirectional	list, [multi]set, multiset, [multi]map
Forward	forward_list, unordered_[multi]{map set}
Output	ostream, inserter
Input	istream

Table 24.2 Which containers and streams provide iterators of which category?

There are some distinctions to make between different categories of iterators, as follows:

- **All iterators**

All iterators can be copied and assigned, meaning you can do `It it(other)` and `it = other`. Every iterator also can be incremented with `++`.

- **Output iterators**

These allow you to assign to their dereferenced element with `*it = ...`.

- **Input iterators**

These support comparison with `it == ij` and `it != ij` and can return the value of the dereferenced element with `*it` and `it->...`

- **Forward iterators**

These are input iterators that in addition guarantee that you can traverse a sequence multiple times if the underlying container does not change. This is the *multipass guarantee*. What that means becomes clear when you consider what happens with iterators that are input iterators but not forward iterators. This is the case, for example, when an iterator reads from `cin`—which changes the state of `cin` upon input and is therefore certainly not multipass capable.

- **Bidirectional iterators**

These are forward iterators, but can also be decremented with `--it` and `it--`.

- **Random access iterators**

Random access iterators are bidirectional and support addition `+=` and subtraction `-/=` with integers, as well as comparisons with `>`, `<`, `>=`, and `<=`. The combination of addition and dereferencing also implies that dereferencing with an offset `it[...]` is supported.

■ Contiguous iterators

Contiguous iterators are random access and guarantee that the elements are arranged contiguously in memory. For example, you can pass a block of elements to a C function by memory address.

In [Table 24.2](#), you can see which iterator categories are provided by which containers or streams.

In the ranges library, there is a corresponding concept for each of these iterator categories. The `forward_iterator` corresponds to the `ranges::forward_range` range concept and so on.

24.2.1 Iterators from Containers

Each container has two different iterator types that you can obtain using various methods:

■ Iterator

This allows you to modify the value in the container upon dereferencing—for example, `*it = 12;`.

■ `const_iterator`

This allows you read-only access to the elements—for example, for `cout << *it`.

Depending on what you intend to do with the iterator, you should use either one type or the other. There are two variants of the `begin()` and `end()` methods. For the Container container, they would look like this:

```
struct Container {  
    iterator      begin();  
    iterator      end();  
    const_iterator begin() const;  
    const_iterator end() const;  
    const_iterator cbegin() const;  
    const_iterator cend() const;  
};
```

If your container is, for example, a `const` parameter of a function, you can only get back a `const_iterator`. If the container is mutable, you should decide which iterator type you need.

Do you not always want to have to type “`std::vector<std::string>::const_iterator`” when programming a loop? You don't have to. `auto` helps you here; the compiler uses it to calculate the correct type:

```
// https://godbolt.org/z/dxb98rWzo  
#include <iostream>           // cout  
#include <vector>
```

```

using std::vector;
vector<int> createData(size_t sz) {
    return vector<int>(sz);           // sz x null
}
void fibonacci(vector<int> &data) {
    for(auto it = begin(data)+2; it != end(data); ++it) { // iterator it
        *it = *(it-1) + *(it-2);
    }
}
std::ostream& show(std::ostream &os, const vector<int> &data) {
    for(auto it=begin(data); it != end(data); ++it)      // const_iterator it
        std::cout << *it << " ";
    return os;
}
int main() {
    vector<int> data = createData(10);
    data[0] = 1;
    data[1] = 1;
    fibonacci(data);
    show(std::cout, data) << "\n";
}

```

Listing 24.8 For “const” objects, “begin()” and “end()” return a “const_iterator”.

In `fibonacci()`, the parameter `data` is not constant. Therefore, `begin()` and `end()` return an iterator of type `vector<int>::iterator`. Thanks to `auto`, you don't need to type this for it yourself; the compiler does it for you.

In `show()`, the parameter `data` is constant, and in that case, you get a `vector<int>::const_iterator` from `begin()` and `end()`—well-suited for read-only output.

24.2.2 More Functionality with Iterators

Many methods of the standard containers do not take an index or pointer as a parameter when referring to an element. After all, it is only `vector` and `array` that know the concept of an index with an integer. Therefore, throughout the entire standard library, you need to specify iterators for positions. Do you want to delete an element from a `vector`, for example? Try this:

```

vector<int> data { 2,5,99,8,3, };
data.erase( data.begin()+2 ); // deletes 99

```

This is even more common for ranges. You can delete everything from 5 to 8:

```

vector<int> data { 2,5,99,8,3, };
data.erase( data.begin()+1, data.begin()+4 ); // deletes 5,99,8

```

It should be noted that the end of the range *always* points to the element *after* the last element of the range. `data.begin()+4` refers to the 3 in `data`; therefore, deletion occurs from 5 up to *before* 3. This applies to all methods of all standard containers and free functions with iterators, with the exception of some methods of `forward_list`.

This is also the reason that the `end()`s of all containers point *behind* the last element. Thus, they can be used easily in algorithms for range specification.

There are many algorithms. In the `<algorithm>` and `<numeric>` headers, you will find very, very many of them. For example, you can sort almost any standard container by calling (essentially) the following:

```
#include <algorithm>
int main() {
    // ...
    std::sort(begin(container), end(container));
    // or with ranges:
    std::ranges::sort(begin(container), end(container));
}
```

With `map` or `set` you don't need to try that, because they are already sorted. But `list`, `vector`, and `array` can be sorted this way.

Finally, you can extend iterators and equip them with new functionalities. For example, in the next listing, see how you can use an `ostream_iterator` output iterator to output the contents of any container to `cout`.

```
// https://godbolt.org/z/7xYcqgh68
#include <vector>
#include <iostream>    // cout
#include <iterator>    // ostream_iterator
#include <algorithm>    // copy

int main() {
    std::vector data { 1, 2, 3, 7, 9, 10 };
    std::ostream_iterator<int> out_it (std::cout, " "); // when assigned to cout
    std::copy(data.begin(), data.end(), out_it); // all elements to the iterator
    std::cout << "\n";                          // Output: 1, 2, 3, 7, 9, 10,
    // or from C++20 with Ranges:
    std::ranges::copy(data, out_it);
    std::cout << "\n";                          // Output: 1, 2, 3, 7, 9, 10,
}
```

Listing 24.9 Iterator adapters change the behavior of operations.

Every time the iterator `out_it` of type `ostream_iterator` is dereferenced and then assigned something, it outputs it to `cout`. Between the elements, it prints a comma with a space ", ". So a `*out_it = 42` results in the output `42, .` The `copy` algorithm from the `<algorithm>` header now performs exactly this assignment for all elements within the range between `data.begin()` and `data.end()`. As a result, you get to see all elements on `cout`.

In the `<iterator>` header, there are other useful things for iterators. For example, you can reverse a range with `reverse_iterator` or use `make_move_iterator` to turn a copying algorithm into a moving one. For the exact functionalities, I refer you to the standard library reference, e.g., <https://cppreference.com>.

24.3 Allocators: Memory Issues

Containers constantly request new memory to store their elements. This can be a complex task, especially for noncontiguous containers like `list` or `set`. To give users the ability to intervene, each container is equipped with an allocator. If you do not specify one, it defaults to `std::allocator<Elem>`.

Allocators Are an Expert Topic

I recommend that you use `std::allocator` and *do not* implement your own allocator. If you do, use ones from the Boost project or other major providers. The concept of allocators implemented in the current standard library is much debated and does not make all experts happy. You should not bother delving deeply into this subject unless you really have to.

To show you how to use allocators, I will now define a very simple allocator. There are much more sophisticated ones, both for general and specific situations.

```
// https://godbolt.org/z/3dYhh6n5f
#include <set>
#include <vector>
#include <iostream>

template<class T> class GobAllocator {
public:
    using value_type = T;
    T* allocate(size_t count) {
        size_t add = sizeof(T)*count;
        std::cout << "allocate("<<add<<"/"<<(buf_.size()-current_)<<")\n";
        if(current_+add > buf_.size()) throw std::bad_alloc{};
```



```
        char* result = buf_.data()+current_;
        current_ += add;
        return reinterpret_cast<T*>(result);
    }

    void deallocate(T* p, size_t count) {
        size_t del = sizeof(T)*count;
        std::cout << "deallocate("<<del<<")\n";
        if(del==current_ && p==reinterpret_cast<T*>(buf_.data())) {
            std::cout << "...all free.\n";
            current_ = 0;                // release everything again
        }
    }

    GobAllocator() : GobAllocator{1024} {}
    explicit GobAllocator(size_t mx)
        : buf_(mx, 0), current_{0} { }

private:
    std::vector<char> buf_;
    size_t current_;
};

int main() {
    constexpr size_t CNT = 1*1000*1000;
    using Gob = GobAllocator<int>;
    try {
        Gob gob(CNT*sizeof(int));        // prepare allocator
        std::vector<int,Gob> data(gob);
        data.reserve(CNT);                // get memory in one go
        for(int val=0; val < (int)CNT; ++val)
            data.push_back(val);
    } catch(std::bad_alloc &ex) {
        std::cout << "Memory exhausted.\n";
    }
}
```

Listing 24.10 A (too) simple allocator and its usage.

The `GobAllocator` acquires a `vector<char>` as a memory reservoir upon creation. It can allocate memory when the using container requests it via `allocate()`, but it rarely deallocates it. The container requests this via `deallocate`. Only if `deallocate()` wants to release all the requested memory will `GobAllocator` actually do something and releases all its memory for new requests.

Using `current_`, the `GobAllocator` keeps track of how much memory it has already managed. When it is asked to allocate memory for `count` objects via `allocate()`, it simply increases the `current_` counter by the number of bytes these objects need. The old position is returned as data storage and converted into a raw pointer of the desired type. This means the container stores its elements here at this location.

It is also important to note that the allocator throws `bad_alloc` as an exception if it fails to obtain the requested memory. This can then be caught using `try/catch`. For example, omit `data.reserve(ANZ);` in [Listing 24.10](#) and observe how the `GobAllocator` runs out of space: the vector now wants to grow continuously and alternates between `allocate()` and `deallocate()` calls to request increasingly larger memory and release the old memory. However, this dumb allocator cannot do that and must eventually give up.

What this could theoretically be used for is reusing memory that has been obtained once.

Here, the memory from `gob` would be used twice completely, but only under the condition that `data1` or `data2` do not grow beyond `CNT` elements:

```
Gob gob(CNT*sizeof(int)); // prepare common allocator
{
    std::vector<int,Gob> data1(gob);
    data1.reserve(CNT);
    // ...
}
{
    std::vector<int,Gob> data2(gob);
    data2.reserve(CNT);
    // ...
}
```

All in all, this simple allocator is quite useless in practice. But at least it shows you how to equip a container with an allocator.

All Containers Support Allocators

All containers of the standard library use an allocator. If you do not specify one, the standard library uses `std::allocator<Elem>`. You can specify your own by passing its type as a template parameter. In addition, you can optionally pass an instance of your allocator to the constructor or let the container create a new one if you do not want to.

The classical allocators described here have a weakness: two containers that are essentially the same but have different allocators are of different types, because an allocator is part of the container type. This means in practice that you cannot assign between them.

The solution is containers with *polymorphic allocators*. Two containers that use these are of the same type even if they use different allocators. Therefore, since C++17, *all containers* are also available in the `std::pmr` namespace—for example, `std::pmr::vector` (*polymorphic memory resource*). Here, you do not specify the allocator with the type but pass it as an additional constructor parameter.

Ahead, I will omit mentioning the allocator as a template argument and constructor parameter for all containers, as well as the mention of the `pmr` variant.

24.4 Container Commonalities

In [Chapter 6](#), [Listing 6.12](#), I used iterators for a loop over the contents of a vector. This brings me to the commonalities of all containers. If you simply replace `std::vector` with `std::list`, `std::set`, or use `std::array<int,5>`, then the rest of the program remains the same. This would not have been possible with `set`, for example, if you had accessed it with an index `[idx]`.

The range-based `for` loop also works internally on iterators and is therefore also possible on all containers. However, it can only ever work on the entire container, whereas an iterator that you obtained with `begin()` is almost arbitrarily manipulable. You will see many examples of this throughout this chapter. As a reminder: I use *italic code* for features from C++23 that might be too recent for some compilers (see [Chapter 1](#), [Listing 1.1](#)).

When designing this part of the standard library, great care was taken to provide a common interface for the containers where it makes sense and works. I do not list the special cases here—they are explained with the specific container—but only give a rough overview:

- **`begin()` `end()`**

All containers offer these two functions to get iterators. Among other things, this allows you to use them in the range-based `for` loop, because they also exist as free functions.

- **`size()` `empty()` `std::ssize()`**

Get the number of elements in any container. Since C++17, they also exist as free functions, and from C++20, there is the free function `ssize()`, which returns `size()` as signed. This avoids warnings about comparisons between signed and unsigned in statements like `for (int i=0; i<cont.size(); ++i)`. Just write `i<ssize(cont)`.

- **`resize()` `reserve()` `clear()`**

You can resize or shrink sequence-based containers (except `array`) with a single call. `vector` and the `unordered` containers can be preinformed with `reserve` about how many elements you intend to store approximately. This saves administrative effort later when actually inserting.

- **operator[] at() data()**
Read and write at arbitrary positions of a container with `cont[where]` or `cont.at(where)`. Not all containers support this. Sequence-based ones require a number for `where`, while associative ones need the data type you chose as the key. With `data()`, you can access the contiguous memory of elements in some containers. This is useful for C functions, among other things. `data()` is also available as a free function.
- **insert() insert_range() emplace()**
Use these to insert or delete elements at arbitrary positions. Note that `vector` can do this, but it is slow. With `insert()`, the element already exists and is copied (or moved in the case of temp-values), while with `emplace()`, it is created in place within the container. The range variant is available from C++23.
- **push_front() push_back() emplace_front() emplace_back()**
Use these to add individual elements to sequence-based containers. Here, a `vector` is also fast.
- **assign() assign_range() swap() merge()**
`assign()` allows you to reinitialize a container. With `swap()`, you can very efficiently exchange the entire contents of two similar containers. `Merge()` combines two similar and presorted containers. `assign_range()` is available from C++23.
- **prepend_range() append_range()**
Since C++23, you can also insert entire ranges into sequence-based containers.
- **erase() extract()**
Remove individual elements or ranges with `erase()`. With `extract()`, you remove an element with the intention of inserting it elsewhere.
- **find() count() contains()**
Find a specific element in an associative container or count how often it occurs. As of C++20, there is also `contains`.

Methods like `c.size()`, which are also available as free functions like `size(c)`, allow for more generic code. Instead of having to modify an existing container type (perhaps from a third party), you can overload the free function and implement the functionality there.

24.5 An Overview of the Standard Container Classes

First, the containers can be roughly divided into four groups:

- **Sequence-based containers**
Access and insertion in these containers occur either in order or element-wise with an index as a number. `vector` and `array` are examples. Typically, you delete and add

elements at one or both ends. Some support efficient direct element access with `[int]` or `at(int)`.

- **Associative ordered containers**

Instead of numbers, these use any data type for access. For example, associate the name of a city with its historical German postal code by writing `cities["Berlin"s] = 1000`. In such containers, you can insert elements at arbitrary positions. The associative containers are kept permanently sorted by their keys. Like sequence-based containers, you can read all elements in order—and because the keys are always kept sorted, you get them delivered in sorted order when iterating. All operations on associative containers have a guaranteed upper bound for resource consumption, regardless of the data you feed into the container.

- **Associative unordered containers**

From the outside, associative unordered containers look like normal associative containers and have almost the same interface. The difference, however, is that their most important operations are usually faster, but not guaranteed. They are based on *hashing* instead of sorting. This hashing can *degrade* and significantly reduce performance. I recommend using ordered associative containers in normal cases and only consciously switching to unordered variants for special cases.

- **Container adapters**

The existing sequence-based and associative containers are so powerful and equipped with such rich interfaces and good performance guarantees that you could also use them for simpler containers. For example, `vector` has all the methods that a stack needs. It would be unnecessary to offer a separate container for `stack` that does not support iteration. For this purpose, there is a small set of *adapters* that internally hold another container but offer a reduced or slightly redesigned interface externally.

24.5.1 Type Aliases of Containers

Ahead, it is sometimes important to know which type we are talking about. A container deals with multiple types. There is of course the container type itself. This consists of a template that becomes an *instantiated* type at the moment you write it with its template arguments. With at least one template argument, you specify the types of the elements that are stored in the container. So, for example, the `vector` container with the `int` element type together becomes `vector<int>` and thus the type of the container.

In addition to data and methods, all containers also contain some type aliases. I have compiled the most important ones in [Table 24.3](#). On the one hand, I will use these terms when I tell you about the containers; on the other hand, you can use the type aliases to write consistent code with different container types—and perhaps later use them in your own templates. In some cases, the type is really just an alias for another more or less simple type; for example, `value_type` of a `vector<int>` is an alias for `int`. In other

cases, like `iterator`, the underlying type is not specified by the standard and is only described by its properties. You must then use the alias because you could not (portably) write the concrete type.

It is not always easy to apply the terms clearly here. For `map`, the term “value” is ambiguous, because it is often used like “it maps a key to a value”. But the `value_type` of `map` is actually together defined as `pair<key_type, mapped_type>`. Therefore, I prefer using the term “target” in that case. I strive to use the following clear terms in the book:

- *Value* or *element* of `value_type`
- *Key* of `key_type`
- *Target* of *target type* or *mapped type* for `mapped_type`

Type Alias	Meaning	<code>vector<int>::...</code>	<code>set<int>::...</code>	<code>map<int, string>::...</code>
<code>value_type</code>	Elements of the container	<code>int</code>	<code>int</code>	<code>pair<const int, string></code>
<code>key_type</code>	Key part of the elements	–	<code>int</code>	<code>int</code>
<code>mapped_type</code>	Mapped part of the elements	–	–	<code>string</code>
<code>size_type</code>	Size or index	<code>size_t</code>	<code>size_t</code>	<code>size_t</code>
<code>reference</code>	Reference to element	<code>int&</code>	<code>int&</code>	<code>pair<const int, string>&</code>
<code>iterator</code>	Iterator in the <i>modifiable</i> container	Yes	Yes	Yes
<code>const_iterator</code>	Iterator in the <i>immutable</i> container	Yes	Yes	Yes
<code>reverse_iterator</code>	Reverse iterator	Yes	Yes	Yes

Table 24.3 Overview of the most important type aliases of containers.

Even if you forgo `auto`, you can use these aliases to write code that looks the same for every container type or create your own templates.

```
// https://godbolt.org/z/c97qra3M7
#include <vector>
#include <map>
#include <iostream>
using std::cout; using std::ostream;
```

```
template<typename K, typename T>
ostream& operator<<(ostream& os, std::pair<const K,T> value) {
    return os << '[' << value.first << ':' << value.second << ']'>;
}

int main() {
    {
        using Cont = std::vector<int>;
        Cont cont{ 1, 2, 3, 4, 5, 6 };
        Cont::size_type sz = cont.size();
        cout << "size=" << sz << " content= ";
        for(Cont::const_iterator it = cont.begin(); it != cont.end(); ++it)
            cout << *it << ' ';
        cout << '\n';
    }
    {
        using Cont = std::map<int,char>;
        Cont cont{ {1,'a'}, {2,'b'}, {3,'c'}, {4,'d'}, {5,'e'}, {6,'f'} };
        Cont::size_type sz = cont.size();
        cout << "size=" << sz << " content= ";
        for(Cont::const_iterator it = cont.begin(); it != cont.end(); ++it)
            cout << *it << ' ';
        cout << '\n';
    }
}
```

Listing 24.11 Type aliases are sometimes clearer than the concrete types.

Here I use the aliases provided by the containers `size_type` and `const_iterator`. In this simple case, `auto` would have worked, but this demonstrates the aliases.

Interlude: “pair” and “tuple”

As shown in [Table 24.3](#), some containers internally use `pair`. As you progress through the chapter, you will encounter several methods that also return a `pair` as a return type. This is nothing more than bundling two data types and their values together—essentially:

```
// https://godbolt.org/z/xcrbvMljz
struct PairIntDouble {
    int first;
    double second;
};

int main() {
    PairIntDouble p{ 2, 7.123 };
}
```

What I have done here specifically for `int` and `double` is possible with `std::pair` as a template class for any types. You can access the first element with `first` and the second with `second`.

A generalization of `pair` for any number of elements is `tuple`. For example, you use `get<3>(t)` to access the fourth element of `t`.

That should suffice for a discussion of `pair` and `tuple` for this chapter. A complete discussion of these two data types can be found in [Chapter 28, Section 28.1](#).

24.6 The Sequential Container Classes

Let's walk through the sequential container classes (see [Table 24.4](#)):

■ `array<ElementType,Size>`

A fixed number of elements is stored here. Adding and removing is only possible by overwriting. The elements are stored directly next to each other in memory. Therefore, `array` is suitable for both a few huge and many tiny element types, as it has virtually no memory overhead.

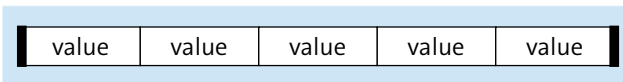


Figure 24.2 An “array” can neither grow nor shrink, but it is compact.

■ `vector<ElementType>`

This all-rounder grows automatically. You can insert anywhere, but efficiently only at the back end.

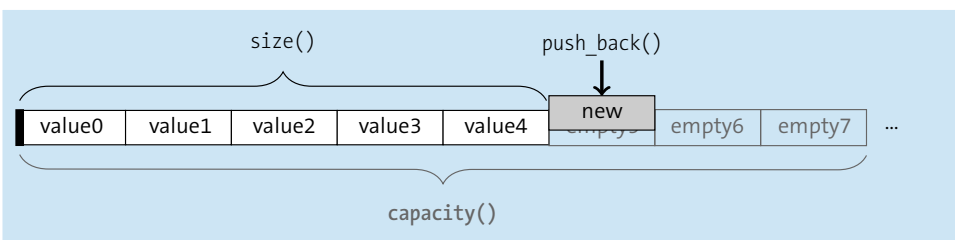


Figure 24.3 Schematic representation of a “vector”.

Reading the elements is done sequentially forwards, backward, or randomly by index. Also, `vector` stores its elements directly next to each other in memory, making it practical for both a few large and many tiny elements. Optimally used, it has virtually no memory overhead; in nonoptimal cases, it is still acceptable as it does not fragment memory.

■ `deque<ElementType>`

You can easily add elements to the end of a vector, but less easily to the beginning, as all existing elements must be shifted one place to the right. The *double-ended queue* deque is suitable for quickly adding elements at both ends. As a disadvantage, it does not store its elements directly next to each other, but still usually more compactly than list. If possible, prefer vector.

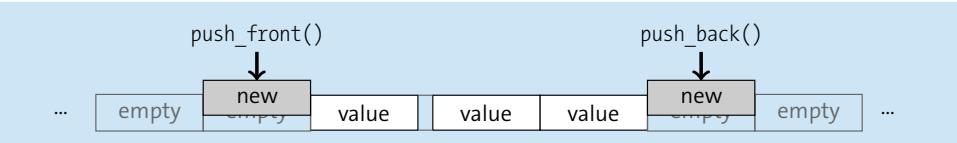


Figure 24.4 A “deque” can be extended at both ends, but the elements are not contiguous in memory.

■ `list<ElementType>`

The list is simpler than a vector and is very well suited for frequent insertions in the middle. However, the elements are not contiguous in memory but are internally linked. You do not have index access available. If you want to iterate over ranges, it is fast.

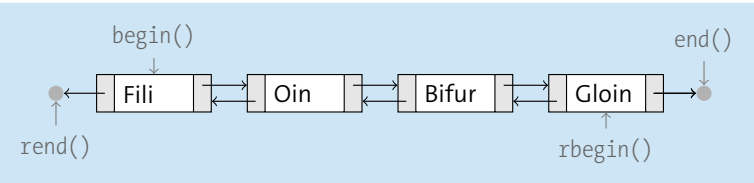


Figure 24.5 A “list” can be traversed forwards and backward. Insertion is easy.

■ `forward_list<ElementType>`

This exotic type mimics the fundamental structure of a linked list from C and therefore requires minimal additional resources. However, it has a slightly different interface than all other containers. Even insertion works somewhat differently. You can only iterate forwards. Both list and forward_list are specialists in *splicing*—that is, transferring some consecutive or all elements from one list to another.

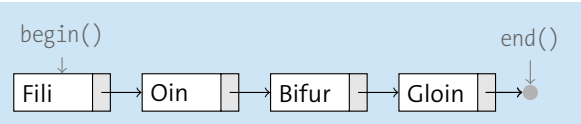


Figure 24.6 Elements of the “forward_list” only know their successor.

Container	Description
array	Fixed size; similar to C-array <i>ElementType</i> [<i>Size</i>]
vector	All-rounder; similar to new <i>ElementType</i> [<i>Size</i>], but dynamically growing; inserting and removing at the back is very efficient
deque	Inserting and removing at the front and back is very efficient
list	Inserting everywhere is efficient; no [<i>i</i>]; forward and backward iterable
forward_list	Efficient insertion everywhere; low memory overhead; no [<i>i</i>]; no <i>size()</i> ; only forward iterable

Table 24.4 Profiles of the sequence containers.

24.6.1 Commonalities and Differences

Among the sequence-based container classes in C++, `vector` and `array` are the ubiquitous all-rounders. If you are considering using a sequence container, one of these two should always be your first choice.

Property	array	vector	deque	list	forward_list
Dynamic size	–	Yes	Yes	Yes	Yes
Key type*	size_t	size_t	size_t	–	–
Iterate forward	Yes	Yes	Yes	Yes	Yes
Iterate backward	Yes	Yes	Yes	Yes	–
Overhead per element	None	Special*	Very Little	Yes	Little
Efficient insertion	Not at all	Back	Front/back	Everywhere	Special*
Where to insert	–	Everywhere	Everywhere	Everywhere	Everywhere
Splicing	–	–	–	Yes	Yes
Memory layout	Stack	Contiguous	Open	Fragm.	Fragm.
Iterators/ranges	Contiguous	Contiguous	Random access	Bidirectional	Forward
Algorithms	All	All	All	Special*	Special*

Table 24.5 Properties of sequence containers. For the * entries, see the explanation ahead.

You use `array` when you know at compile time how many elements the container should hold. `vector`, on the other hand, has a dynamic size: you can add elements to it or remove them.

[Table 24.5](#) lists the main properties that distinguish the sequence-based containers from each other. The following notes apply to the entries marked with an asterisk *:

- **Key type**

You can only use a `size_t` for `operator[]` and `at`. *Key* is not quite the right term here, because shifting elements changes the association of the keys to the container elements. However, I include this property here to compare it with associative containers.

- **Overhead per element of vector**

In the optimal case, `vector` has virtually no additional memory consumption other than that of the bare elements it manages. If you know in advance that you will store 1,000 elements, you can prepare the vector to exactly this size with `reserve(1000)` and then add elements. You will achieve optimal performance in terms of time and memory consumption. If you do not reserve a size in advance, the vector will be slightly larger than needed, and new space will have to be allocated from time to time when adding elements. However, the internal algorithms ensure that, on average, you spend at most twice as much time or memory as without the advanced reservation.

- **Efficient insertion into the `forward_list`**

Unlike all other containers, when you insert into the `forward_list`, you do not insert at the position you indicate, but rather *one element after*. Why this is the case will be shown later in an example. For this reason, this container, unlike any other (except `array`), does not have `insert` but instead has `insert_after`. This inconvenience is due to the desire to use as little memory per element as possible.

- **Splicing of `list` and `forward_list`**

These two containers have the unique property compared to all others that you can merge two containers of the same type extremely quickly. Making one out of two does not cost a single copy operation of the elements contained within these two, and the administrative information to be changed is independent of the number of elements in the lists.

- **Memory layout**

On many occasions, the elements of the container need to be packed closely together to work with them optimally. For example, when writing to the hard drive, it is advantageous to pass all the data to be written to the operating system as one large block. And other C functions also struggle with complex C++ data structures. They require a starting point in memory and a length (hello, `span!`). Of all the containers, only `array` and `vector` store their elements in such a way that this type of data exchange works, with `array` being the only container that can use only the stack and does not need to request heap space with `new`.

■ Algorithms

The containers that offer random access iterators work with all free functions that take an area in the form of a pair of iterators as parameters. However, lists do not offer random access and refuse to work with algorithms. Special methods that they provide instead solve this problem.

■ Spans

You can access contiguous sequence containers (including C-array) with `span` since C++20. This class contains an iterator to the first element and the number of elements in the container. Both are immutable and set upon creation. Conversions of a container to a span happen implicitly, for example, when passing it to a function that expects a span. For string, there is `string_view` for this purpose. In C++23, `mdspan` is introduced to create a multidimensional view like a matrix or cube from a one-dimensional vector (or similar).

24.6.2 Methods of Sequence Containers

In this section, `Cont` or `C` stands for one of the container types like `vector<int>` or `array<Hobbit,4>`. Here, `cont` stands for an instance of a container, as in `list<double> cont;`. For *element type*, I use `Elem` or `E`, and for *iterator types*, I abbreviate with `It`. `rg` stands for a range.

To save space and increase clarity, I am not always very precise or detailed with the function signatures here. If I do not include the parentheses with `method`, there are usually multiple overloads:

■ Constructors: `C()`, `C(size_t)`, `C(size_t, E e)`, `C{...}`, `C(It, It)`, `C(from_range, rg)`

The default constructor `Cont()` creates an empty container. With `Cont(size_t)`, you can create a container of a predetermined size filled with default-initialized elements, such as `vector<int>(5)`, which creates a vector with the value 0 five times. If you do not like the default value, specify your own: `vector<double>(5, 42.0)` will contain the value 42.0 five times, for example. With a pair of iterators from any other container, you can copy its contents. Since C++23, you can also use the `from_range` marker to initialize from a range. You can specify an initializer list that your container should contain at the beginning. *Exception:* You always create an array using an initializer list.

■ Copy and Move: `C(const C&)`, `C& operator=(const C&)`, `assign`, `assign_range`, `C(C&&)`, `C& operator=(C&&)`, `swap`

You can pass a different `Cont` as an argument to the copy and move constructors and operators. There is an `assign` method in several variants to reinitialize the container similarly to one of the constructor options, with `assign_range()` existing since C++23. All containers implement `swap(Cont& other)` efficiently (in constant time, independent of the number of elements), which allows for great performance tricks

when used skillfully; see [Listing 24.51](#). *Exception:* `array` has no `assign` methods, and `swap` takes linear time for `array`.

■ **Destructor:** `~Cont()`

All containers have destructors that remove the contained elements. Note that this does not apply if you pack raw pointers into containers. Use smart pointers for that, or manage the dynamic objects elsewhere. The destructors do not implicitly throw exceptions.¹

■ **Iterate forward:** `begin()`, `end()` and **backward:** `rbegin()`, `rend()`

You get an iterator with `begin()` that points to the first element, and with `end()` one that points *behind* the last element; you can use these iterators for a `for` loop to iterate forward. For the backward loop, use `rbegin()`, which points to the last element, and `rend()`, which points *before* the first element. In sequence containers, you always iterate over the elements in the order in which you inserted them or where you inserted them. All these methods return a `Cont::iterator` or `Cont::const_iterator`, and they also come in variants with a `c` prefix, such as `cbegin()`, which always returns a `Cont::const_iterator`. *Exception:* `forward_list` has neither `rbegin()` nor `rend()`.

■ **Element access:** `E& operator[]`, `E& at(size_t)`, `E& front()`, `E& back()`

With `cont[i]`, you can access the *i*-th element unchecked. The `at` method first checks whether the container contains the element and throws an `out_of_range` exception if it does not. `c.front()` is equivalent to `*(c.begin())`, and `c.back()` is equivalent to `*(prev(c.end()))`. *Exceptions:* `list` and `forward_list` have neither `at` nor `[]`, and `forward_list` has no `back()`.

■ **Size:** `size()`, `empty()`, `resize(size_t)`, `max_size()`

With `size()`, you get the current number of elements in the container; `empty()` is true if this number is zero. When you call `resize`, the container shrinks or grows and may remove or create elements accordingly. With `max_size()`, you get the theoretical maximum size of the container. Note that this does not consider the current state of the program or the machine but is a library-dependent constant. *Exceptions:* `forward_list` has no `size()`, and `array` has no `resize`.

■ **Capacity:** `capacity()`, `reserve(size_t)`, `shrink_to_fit()`

Only `vector` has a *capacity* different from its size that you can influence. `capacity()` returns the number of elements that `vector` can hold without needing to request new dynamic memory. If you pack more than `capacity` elements into it, this capacity automatically increases. Internally, this results in all elements being copied or moved. This sounds worse than it is, but if you know in advance how many elements you will pack into a `vector`, it is worth requesting this capacity once with `reserve()`. If you do not know this in advance but can estimate when you are done adding elements, you can get rid of excess capacity with a `shrink_to_fit()` call. However, this approach can also trigger an internal copy or move, and `vector` does not have to

¹ If the destructors of the elements do not do it, which they should not.

comply with the request. *Exceptions:* As deque is usually based on vector, it also has a `shrink_to_fit()`.

- **Adding and removing at one end:** `push_back(E)`, `push_front(E)`, `pop_back()`, `pop_front()`, `prepend_range()`, `append_range()`

`push_back` adds an element at the back, `push_front` at the front. The `pop_...` methods remove an element. For vector and deque, you must expect that references and iterators to the container become invalid after these operations. The *range* methods are available from C++23. *Exceptions:* array does not have any of these methods, vector lacks the `front/prepend` methods, and `forward_list` lacks the `back/append` methods.

- **Adding and removing with emplacement:** `emplace_front(...)`, `emplace_back(...)`

While the `push_...` methods take a ready-made element as an argument, which is then copied into the container, `emplace_...` can save you this copy. Simply provide these methods with the desired constructor arguments of `Elem(...)`, and the new object will be created directly in place. *Exceptions:* array lacks these methods, and vector lacks `emplace_front`. The container `forward_list` lacks `emplace_back`.

- **Other modifications:** `clear()`, `erase`, `insert` and `emplace`

`clear()` simply removes all elements from the container, leaving it with a size of zero. With `erase`, you can remove a single element using an iterator or an entire area using a pair. Similarly, you can use `insert` and an iterator as a position to insert something at a specific spot—a single element (or multiple copies of it), the elements of an initializer list, or an area from another container using a pair of iterators. With `emplace`, you can create one or more elements at a specified position in the container without copying. *Exceptions:* array has none of these methods, and `forward_list` instead has `...after()` methods (but does have `clear()`).

- **List operation:** `splice`, `splice_after`, `merge` and `algorithm` methods

Only list and `forward_list` can easily merge ranges of different containers with `splice` or `splice_after`—removing from one and adding to the other. And among the sequence containers, only these two can merge two presorted lists in sorted order using `merge`. Because the two list containers do not offer random access iterators, the free functions from `<algorithm>` are not applicable to them. As a substitute, they offer methods that mimic the algorithms: `merge`, `remove`, `remove_if`, `reverse`, `unique`, and `sort`. Note that unlike their counterparts in `<algorithm>`, the `remove` methods of the lists actually remove the found elements rather than just changing their position.

24.6.3 “vector”

vector should be the first thing that comes to mind when you need a container that must be able to grow and shrink. You can add elements to it—preferably at the back, but elsewhere if necessary. The elements are in the order you specify in the container, so they are not automatically sorted.

Because it guarantees to store its elements in a contiguous block, it is suitable for use with C functions and similar, which receive input or output data in this manner. Here, `vector` works well with the `span` introduced in C++20, as it bundles a pointer to the beginning and the length, as often expected by C APIs. You can determine and pass on the address of an element. Just like the container's iterators, addresses to its elements remain valid as long as you do not change the size or capacity of the container.

The space for the elements of a `vector` is automatically adjusted, increased, or decreased as needed. Space is reserved for upcoming elements so that the size does not need to be adjusted for each individual insertion. This happens in such a clever way that even in the worst case, each insertion on average proceeds at $O(1)$. You can determine how much space the `vector` currently holds with `capacity()`. If you know in advance how many elements will end up in the `vector`, request the memory with `reserve(size_t)` to avoid future expansions.

You can access the elements of the `vector` randomly in $O(1)$ —that is, with optimal speed.

Tip: Locality

For the CPU, it is better to access registers than cache. Cache is better than main memory, and main memory is faster than the hard drive. The more compact the memory areas needed for a computation are, the more likely it is that the computer can use fast memory for it. Even better for locality is if the memory is mainly used in one direction rather than back and forth. Often, the same computation using a strategy with better locality is many times faster: factors of $\times 10$ or $\times 100$ are possible!

A `vector` has better locality than, for example, a `set`, if no special precautions with specific allocators are taken. If you can also use a `vector` in ascending order, then that is even better for locality. To do this, let the iterator only move forward in the `vector` or increment the index for access only.

If you want to keep data always sorted, you usually use a `set`. However, that is not good for memory. If this is important to you, take a look at the `flat_set` adapter for a `vector`, which has been available since C++23.

You can pack almost anything into a `vector` that is possible in C++ types. It depends on the operations you will use and whether the type is allowed or not. The element type must either be copyable or movable; the standard library will choose the appropriate operation. However, there must be a destructor for `Elem`, either one generated by the compiler or one you wrote yourself—so you cannot mark the destructor with `= delete`.

In practice, it is recommended that the element type be movable and guaranteed not to throw exceptions when moved—that is, either marked with `noexcept` move operations or generated by the compiler. If you can, stick to the rule of zero for the element type; see [Chapter 17, Section 17.1](#) as a reminder.

Assume that all the following examples in this section for vector are embedded in the code from [Listing 24.12](#).

```
// https://godbolt.org/z/WK86eaz1x
#include <vector>
#include <iostream>
using std::vector; using std::cout;
template<typename T>
std::ostream& operator<<(std::ostream&os, const vector<T>& data) {
    for(const auto &e : data) {
        os << e << ' ';
    }
    return os;
}
int main() {h
    // Example code here
}
```

Listing 24.12 This is the template for the example listings in this section for vector.

Operator “<<=”

In some examples in this chapter, I use a self-defined `cout <<= ...` instead of `cout << ... << '\n'`. I do this only for brevity in the presentation in this book.

Do not mix the two operators in an expression: `<<=` has a lower operator precedence than `<<` and would be executed last. Use it at most at the end of the food chain, perhaps in tests.

Initialize

I will exemplarily go through the various ways to initialize a vector with you. These are mostly repeated for the other containers, especially the sequence containers. Elsewhere, I will briefly touch on the common initializations and then focus on the differences.

You can initialize a vector with the default constructor.

```
// https://godbolt.org/z/7zMP5fo7c
vector<int> dataA;
vector<int> dataB{};
vector<int> dataC = {}; // no assignment
cout << format("{} {} {}\\n", dataA.size(), dataB.size(), dataC.size()); // 0 0 0
```

Listing 24.13 The default constructor initializes an empty vector.

I have applied the C++20 `format()` here.

You can also copy a vector. Here you can also see that since C++17, the compiler deduces the template parameter `<int>` from the constructor arguments.

```
// https://godbolt.org/z/cE6G1aErs
auto count(vector<int> arg) { return arg.size(); }
// ...
vector input{1,2,3};           // vector<int>
vector outputA(input);         // copy
vector outputB = input;        // also copy, no assignment
cout << count(input) << '\n'; // call by copy-by-value
```

Listing 24.14 Copying a vector using the constructor or implicitly.

Whether automatically by the compiler or intentionally with `std::move`, a vector can also steal the data of another vector. With the help of `make_move_iterator`, you can modify any iterator so that it applies `std::move` to each individual element. Thus, you can move not only all elements of another container but also those of a range.

```
// https://godbolt.org/z/fv5WEe97M
#include <list>
#include <string>
#include <iterator>           // make_move_iterator
using std::make_move_iterator; using std::string;
vector<int> create() { return vector<int>{8, 9, 10}; }
size_t count(vector<int> d) { return d.size(); }
// ...
vector input{1,2,3};
vector outputA(std::move(input)); // move
vector outputB = std::move(input); // also move, not assignment
vector data = create();           // Return-Value-Optimization
cout << count(input) << '\n';     // call by Copy-by-Value
// move elements from another container
std::list<string> source{ "a", "a", "a", "BB", "CC", "DD", "b", "b" };
auto from = source.begin();
std::advance(from, 3); // 3 forward, but slow in list
auto to = from;
std::advance(to, 3); // another 3 forward
vector target(make_move_iterator(from), make_move_iterator(to));
// source is now {"a", "a", "a", "", "", "", "b", "b"}, target is now {"BB", "CC", "DD"}
```

Listing 24.15 For return values, the compiler can often also move.

The `move_iterator` created by `make_move_iterator` ensures that every `a = b` during the element-wise initialization of `target` becomes `a = std::move(b)`. And this means for

string that `b` will contain the empty string afterward. Moving from `source` to `target` for three elements does not move the entire elements, but rather applies a move operation element-wise—meaning that the *content* of the elements is moved, not the element itself.

All containers support initialization with an initializer list and can usually deduce the template parameter for you since C++17. However, avoid the pitfall of putting C string literals into the initializer list. In such a case, explicitly write `vector<string>` to force a conversion or use the `"s"` suffix for `string` (or even `"sv"` for `string_view`).

```
// https://godbolt.org/z/6rG393Wa5
vector<int> primes{ 2,3,5,7,11 };
vector<int> evens{ 2,4,6,8,10 };
vector<int> notLikeThis{ 'a', 4.3, 8L }; // ⚡ "Narrowing" double not okay
vector<string> names{ "are", "only" }; // converts arguments
vector<string> sound{ "smoke", "fume" }; // dangerous: vector<const char*>
vector<string> wet{ "rain"s, "water"s }; // vector<string>
vector<string> cold{ "ice"sv, "pole"sv }; // vector<string_view>
```

Listing 24.16 Using an initializer list to prefill a vector. Pay attention to the correct types in the list.

Narrowing and Initialization Lists

Narrowing is the implicit conversion of a numeric data type `A` into another type `B`, where `B` is not capable of holding the entire range of values of `A`, such as `int` to `char` or `double` to `int`.

In an *initialization list*, the compiler does not allow such conversions to protect you from typical and hard-to-find errors.

In [Listing 24.15](#), the compiler complains that you are trying to convert the `double` value `4.3` into an `int`. For the long value `8L`, it does not complain because it is smart enough to realize that an `int` is sufficient for this. `800000000L` will allow it for a 32-bit `int` but not otherwise.

All standard containers can copy data from other containers when fed with iterators.

```
// https://godbolt.org/z/h8Pze74ME
#include <deque>
#include <ranges> // C++20
// ...
std::deque in{1,2,33,34,35,99};
vector thirty(in.begin()+2, in.begin()+5);
for(auto &e : thirty) {
    cout << e << ' ';
}
```

```
cout << '\n';
namespace vs = std::ranges::views;           // C++20
auto v = in | vs::drop(2) | vs::take(3);
vector otuz(v.begin(), v.end());
vector trento(std::from_range, in);          // C++23
```

Listing 24.17 Copy values from any other container or C-array during initialization.

Since C++20, this is also possible with the iterators of ranges and views, and from C++23 even without iterators, if you additionally pass the *from_range* marker to the constructor.

Specifically for vector, there are constructors that preinitialize a vector with a number of elements.

```
// https://godbolt.org/z/addobjfsK
vector<int> zeros(10);           // 10 zeros
vector<int> sixes(10, 6);        // 10 sixes
vector<int> ten{10};             // ⚡ Attention! Only one 10
vector<int> tenSix{10, 6};       // ⚡ Attention! Two elements 10 and 6
```

Listing 24.18 Preinitializing with a fixed number of values is (almost) only available with “vector”.

In these cases, you must use the round brackets () during initialization; otherwise the elements will be interpreted as an initializer list.

Assignment

Like all containers (except array), vector has overloaded the assignment operator `operator=` for copying and moving. With the help of `assign`, you can usually apply all those initializations to a container even after construction that you can apply during the “first” initialization. Starting from C++23, you can also use *assign_range*.

```
// https://godbolt.org/z/Yb4jEdaKj
vector<int> from{ 2,3,4 };
vector<int> to{};
to = from;                               // Assignment with operator=, now both are the same

vector<int> drain{};
sink = std::move(from);                   // Move, now 'from' is empty
vector<int> v;
v.assign(4, 100);                         // v is now {100, 100, 100, 100}
v.assign(to.begin(), to.end());           // v is now {2,3,4}
```

```
int z[] = { 10, 20, 30, 40 };
v.assign(z+1, z+4);           // v is now {20, 30, 40}
```

Listing 24.19 You can assign to “vector” or reinitialize it later with “assign”.

Accessing a “vector”

Somehow you need to get to the data that is in the `vector`. There are mainly these possibilities:

- **Iterator**

You use an indirection, mainly an *iterator*. Because the elements are contiguous, pointers also work.

- **Index**

You use an *index* to directly address an element. For this purpose, use `at` and `operator[]`.

- **Beginning and end**

`front` and `back` return references to the first and last element. As the result is undefined for an empty container, you must combine the query with, for example, `!empty()` if necessary.

- **Span**

A span is easier to handle than two separate iterators or an iterator and a count. Hence, `span` bundles this into a single entity. This class has been available since C++20, and since C++23, there is also the multidimensional variant `mdspan`. For more on this, see the “Access via Spans” section ahead.

The `vector` container has several member functions that return an iterator, primarily `begin()` and `end()`, as with all containers. And just like with all other containers, you *dereference* an iterator to get a reference to the actual element, as shown in [Listing 24.20](#).

```
// https://godbolt.org/z/freaK1Pea
vector vowels { 'A', 'e', 'i', 'o', 'u' };
const vector even { '0', '2', '4', '6', '8' };
auto it1 = vowels.begin();           // vector<char>::iterator
*it1 = 'a';                         // '*it1' returns 'char&'
auto it2 = even.begin();             // vector<char>::const_iterator
auto it3 = vowels.cbegin();          // enforces const_iterator
*i2 = '9'; *i3 = 'x';               // ⚡ 'const char&' is not modifiable
for(auto it=vowels.cbegin()+1; it!=vowels.cend(); ++it)
    { cout << *it; } cout << '\n';  // Output: eiou
for(auto it=vowels.crbegin()+1; it!=vowels.crend(); ++it) // ++ despite reverse!
    { cout << *it; } cout << '\n';  // Output: oiea
```

Listing 24.20 With “begin”, “end”, and their relatives, you get iterators.

With the variants `cbegin` and `cend`, you always get a `const_iterator` back, which does not allow you to modify the element upon dereferencing. `begin` and `end`, on the other hand, return either a `const_iterator` or an `iterator` depending on the container's modifier.

The `r...` variants of these functions allow reverse iteration through the vector. Note that you need to increment `reverse_iterator` or `const_reverse_iterator` with `++` to actually iterate backward.

Notation for “`begin()`”, “`cbegin()`”, “`rbegin()`”, and “`crbegin()`”

To be more concise, I write

- `[c]begin()` for `begin()` and `cbegin()`;
- `[r]begin()` for `begin()` and `rbegin()`; and
- `[cr]begin()` for `begin()`, `cbegin()`, `rbegin()`, and `crbegin()`.

The same applies to the variants of `end()`.

Access via iterator is universal for all containers. `vector` has iterators with random access—that is, via `random_access_iterator_tag`. In practice, this means that you can add a positive or negative offset to a vector iterator, as you can see in [Listing 24.21](#).

```
// https://godbolt.org/z/srczx737G
#include <vector>
#include <iostream>
#include <algorithm>           // ranges::sort
using std::vector; using std::cout;
double median(vector<int> data) { // copied
    std::ranges::sort(data);      // C++20, otherwise std::sort()
    auto it = data.begin();
    auto sz = data.size();
    if(sz==0) return 0;           // special case
    // Determine median:
    auto m = (it+sz/2);           // approximately the middle
    if(sz%2 != 0) {               // odd number of elements
        return *m;
    } else {                     // even number of elements
        return double(*m + *(m-1)) / 2;
    }
}

int main() {
    vector data1 { 12, 22, 34, 10, 1, 99, 33 };
    cout << median(data1) << '\n'; // 22
    vector data2 { 30, 2, 80, 99, 31, 3 };
}
```

```
    cout << median(data2) << '\n'; // 30.5
}
```

Listing 24.21 With “vector” iterators, you can perform random access.

`it` points to the beginning of the vector. With $m = it + sz/2$, you move forward by half the vector, and `m` points approximately to the middle. For an odd number of elements like 7, $sz/2$ naturally results in 3, and `it+3` corresponds to `data[3]`, which is the middle of 7 elements. If the number is even like 6, then the middle lies between the elements `data[2]` and `data[3]`. `m` points to `data[3]`, and thus $(*m + *(m-1)) / 2$ calculates the average of the surrounding values. To ensure that the division also occurs with floating-point numbers, I convert the intermediate result beforehand with `double(...)`.

While every container works with iterators, only `vector`, `array`, and `deque` allow access to the integer position in the container using `at` and `operator[]`. (Of course, that’s aside from the fact that you can also use an associative container with integers as keys.)

The frontmost element always has the index zero, and the last in the container has the index `size()-1`:

■ Unchecked access

When you access a vector with `Elem& operator[](size_t idx)`, you immediately get back the element at the stored position—that is, in $O(1)$. It is not allowed to access an element beyond `size()-1` of the vector. That would be a programming error and would lead to undefined behavior. You must ensure that this does not happen in your program.

■ Checked access

`Elem& at(size_t idx)` is very fast, in $O(1)$, but the library checks at runtime whether you are using an index within the allowed range of the container. If not, then it is not a programming error, but it throws an exception of type `std::out_of_range`.

You can use `[]` and `at` both for reading and on the left side of an assignment for writing the element.

In a loop, it is smart to avoid `at` and use `[]`, because typically the loop condition already includes a check.

```
// https://godbolt.org/z/jjWx3d375
vector d{ 1, 2, 4, -1, 1, 2, -2 };
for(size_t idx=0; idx < d.size(); ) { // checks vector boundary
    cout << d[idx] << ' ';           // additional check with at not necessary
    idx += d[idx];                   // same here
}
cout << '\n';
// Output: 1 2 -1 4 -2 1 2
```

Listing 24.22 Normally you access with `[]`, because you already check the boundaries elsewhere.

Here, the loop's termination condition `idx < d.size()` already checks if access in the loop body is allowed. Using `d.at(idx)` would be wasted effort.

For const vector, both `at` and `[]` naturally return `const Elem&`.

```
// https://godbolt.org/z/jGx51caTj
#include <vector>
#include <iostream>
void printAndMore(const std::vector<int>& data) { // by-const-ref
    std::cout << data[0] << std::endl;
    data[0] = 666; // ⚡ doesn't work because 'const int&'
}
int main() {
    std::vector numbers {1,2,3};
    printAndMore(numbers);
}
```

Listing 24.23 Constant container returns constant reference.

Unlike associative containers, the position of elements changes when you delete or insert items.

```
// https://godbolt.org/z/s9PIY6jhv
vector<string> cars{ "Diesel", "Petrol", "Super", "Gas" };
cout << cars[1] << '\n'; // Output: Petrol
cars.insert(cars.begin(), "Electricity"); // shifts everything back by one
cout << cars[1] << '\n'; // Output: Diesel
```

Listing 24.24 “insert” shifts all elements back by one here.

By using `insert` at the beginning of the vector, all data is shifted back by one position, and the content of `cars[1]` changes. An `insert` anywhere in the vector takes relatively long with $O(n)$, and you should use it sparingly. Prefer `push_back` and `emplace_back`.

`vector` and `array` are the only containers with the `data()` method, which returns a *raw pointer* to the first element. And because these two guarantee that the contained elements are stored *contiguously*, you can use this pointer to access *all* elements as if it were a C-array. Therefore, `data` is well-suited for passing data to a C function, as you can see in [Listing 24.25](#) with the example of `fread` and `fwrite`. With C++20, it is better to use `span` to keep pointers and lengths together, as shown in [Listing 24.25](#).

```
// https://godbolt.org/z/Kx1KcP65T
#include <vector>
#include <iostream>
#include <cstdio> // fopen, fclose, fwrite, fread, remove
```

```

using namespace std;
ostream& operator<<(ostream&os, const vector<int>&data) {
    for(auto &e : data) os << e << ' '; return os;
}
static const char* FILENAME = "nums.dat";
int main() {
    const vector nums{10,11,22,34};
    { // write
        auto out = fopen(FILENAME, "wb"); // Open file with C for writing
        if(out==nullptr) {
            cerr << "Error opening file\n"; return -1;
        }
        auto ok = fwrite(nums.data(), sizeof(int), nums.size(), out);
        if(ok!=nums.size()) {
            cerr << "Error writing to file\n"; return -1;
        }
        fclose(out); // In C, you must explicitly close. Honestly.
    }
    vector<int> read{};
    { // read
        auto in = fopen(FILENAME, "rb"); // Open file with C for reading
        if(in==nullptr) {
            cerr << "Error opening file\n"; return -1;
        }
        const size_t sz = 4; // assumed, we know we are reading 4 elements ...
        read.resize(sz); // make space for data to be read
        auto ok = fread(read.data(), sizeof(int), sz, in);
        if(ok!=sz) {
            cerr << "Error reading\n"; return -1;
        }
        fclose(in);
    }
    { // Compare
        cout << nums << '\n'; // 10 11 22 34
        cout << read << '\n'; // 10 11 22 34
    }
    if(remove(FILENAME) == -1) {
        cerr << "Warning: Error deleting\n";
    }
}

```

Listing 24.25 Use `data()` as an interface to C.

In [Listing 24.25](#), I use the C functions `fread` and `fwrite` from the `<stdio>` header of the C standard library. These each take a `void*` for the data as the first parameter and then two additional parameters to calculate the size of the data to be read: first the size of each individual element, here `sizeof(int)`, and second the number of `int`. When reading, I make it easy for myself here and assume that I know I want to read four `int` values. In real life, you would probably have to find out differently how many elements you want to read.

Note that the `data()` pointer (as is also the case for iterators) is only valid until you add an element to the vector and thereby change its capacity.

Access via Spans

An extension—that is, `span`—serves as access to a vector (or an array). The previous section provided an overview of this. The `span` class has been available since C++20 and is so useful that I would like to go into more detail about it in this separate section.

A typical use case is that you want to pass a container as a parameter to a function. Without `span`, it would look like this:

```
// very specific parameter type
void func(vector<int>& v);
```

But that unnecessarily restricts to a specific container type and the range of the entire container. It doesn't cost much more to make the parameters iterators and thus remove the restrictions on the range:

```
// general
void func(vector<int>::iterator b, vector<int>::iterator e);
```

Or directly with C++20 concepts to avoid unnecessary coupling with `vector`:

```
// C++20 concept
void func(contiguous_iterator auto b, contiguous_iterator auto e);
```

Now there are two parameters, which is error-prone and not nice. With `span`, the two separate parameters are now combined into one:

```
// C++20: span
void func(span<int> area);
```

The `span` thus combines the two iterators into a single entity. Besides the fact that fewer entities are always better for the programmer, it also protects against some typical errors. Isn't that neat! The next listing shows how you use it.

```
// https://godbolt.org/z/h7jf4KM1q
#include <vector>
#include <span>
```

```

#include <iostream>
using namespace std;
int sum(span<const int> area) { // C++20: span
    int sum = 0;
    for(auto e : range) {      // algorithm would be better ...
        sum += e;
    }
    return sum;
}
int main() {
    vector data {1,2,3,4,5};
    cout << sum(data) << "\n"; // implicit container to span
    auto [b, e, sz] = make_tuple(begin(data), end(data), size(data));
    cout << sum(span{b,    sz-1}) << "\n"; // 1.4 (Iter, size)
    cout << sum(span{b+1, sz-1}) << "\n"; // 2..5 (Iter, size)
    cout << sum(span{b,    e-2}) << "\n"; // 1..3 (Iter, Iter)
    cout << sum(span{b+1, e-1}) << "\n"; // 2..4 (Iter, Iter)
    cout << sum(span{b+2, e  }) << "\n"; // 3..5 (Iter, Iter)
    return 0;
}

```

Listing 24.26 How to use span.

Here you see several ways to create a span:

- Implicitly from a container, such as when it is passed as a parameter
- Explicitly from an iterator and a size
- Explicitly from two iterators

Note that the `std::span<int> area` parameter is not passed by reference, but by value, because it is just a small object.

Other ways to create a span include from a C-array, from a range, and of course from another span. But it is also useful the other way around: say that you have a C++ object and want to pass it to a C API. In the earlier [Listing 24.24](#) with `fwrite`, it would have looked like this:

```

auto write(span<const int> what, FILE* where) { // C++20: span
    return fwrite(what.data(), sizeof(int), what.size(), where);
}

```

A span has an optional template argument `Extent` that specifies the size of the span at compile time. If you omit it, span is dynamic as shown in [Listing 24.26](#). You cannot change the size of the extent directly, but only in relation to the underlying container:

```
span sp{vec.data(), 3};           // the first three elements
sp = span{vec.end()-5, vec.end()}; // Assignment, now the last five elements
```

You create fixed-size extents, for example, as in the next listing.

```
// https://godbolt.org/z/a3qqr1YPM
int car[5] {1,2,3,4,5};
span span_1 = car;           // directly from a C-array
array arr {1,2,3,4,5};
span span_2 {arr};          // directly from a std::array
vector vec {1,2,3,4,5};
span<int,3> span_3 {vec};     // with 'Extent' from a std::vector
```

Listing 24.27 These are spans with fixed sizes.

You can use fixed-size spans when you want to take advantage of the compiler performing certain size checks at compile time for you:

```
std::array arr{1, 2, 3, 4, 5, 6}; // size 6
std::span<int, 3> sp3{vec};       // Size 3
sp3 = arr;                        // ✗ does not compile
```

The `as_bytes(span)` and `as_writable_bytes(span)` free functions are also very exciting. With these you can convert a span into `span<const byte>` or `span<byte>`. You could also use these here for `fwrite()` or even change the underlying memory location.

An example of how you can write back into the underlying container through an extension is shown in the next listing.

```
// https://godbolt.org/z/6Ez5GdozG
#include <vector>
#include <span>
#include <iostream>
using namespace std;
void inc(span<int> span) {
    for(auto& e :span) { // Reference
        e += 1;         // write
    }
}
int main() {
    vector data {1,2,3,4,5};
    span whole{data};    // 1,2,3,4,5
    inc(whole.first(3));  // -> 2,3,4,4,5
    inc(whole.last(3));   // -> 2,3,5,5,6
    inc(whole.last(4).first(3)); // -> 2,4,6,6,6
    inc(whole.subspan(1,3)); // -> 2,5,7,7,6
```

```

    for(auto i: whole) cout << i << ' '; cout << '\n'; // Output: 2 5 7 7 6
    return 0;
}

```

Listing 24.28 You can also write through a “span”.

Here, `inc()` writes through the span into the container. In addition, you see other ways to create a span with `first`, `last`, and `subspan`.

span also offers the `begin()` and `end()` container methods as well as `front()` and `back()` and also operator `[]`.

Dangerous Uses of “span”

You must be careful, as with all references, that the underlying data is not destroyed while you are using span. Be particularly careful when dealing with temp-values:

```

vector<int> create_data() { return {1,2,3,4,5}; }
span<int,5> sp{create_data()};           // ⚡ sp references a temporary object
sp_ = create_data();                     // ⚡ this also references a temp-value
for (auto e: span{create_data()}.last(3)); // ⚡ references a temp-value
auto data = create_data();
span ref_data{data};                     // okay, data is not a temp-value
data.push_back(6);                       // may trigger a reallocation
cout << ref_data[0] << '\n';             // ⚡ possibly invalid access

```

The *mdspan* (*multidimensional span*) added in C++23 works differently. You can overlay it on a one-dimensional data range, and the *mdspan* calculates the indices for you so that you can use them like a multidimensional array. Together with another feature added in C++23, with which operator `[]` can now receive multidimensional indices, it looks as shown in the next listing.

```

// https://godbolt.org/z/n8jn7v5vb
#include <mdspan>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    // 1D: 12 elements
    vector v{1,2,3,4,5,6,7,8,9,10,11,12};
    // 2D: as 2 rows with 6 ints each
    auto ms2 = mdspan(v.data(), 2, 6);
    // 3D: as a cuboid with 2 layers, 3 rows, 2 columns
    auto ms3 = mdspan(v.data(), 2, 3, 2);
}

```

```

// write via 2D view
for (auto i = 0; i != ms2.extent(0); ++i)
    for (auto j = 0; j != ms2.extent(1); ++j)
        ms2[i, j] = i * 100 + j; // write via multidimensional index
// read via 3D view
for (auto i = 0; i != ms3.extent(0); ++i) {
    cout << "Level " << i << ":\n";
    for (auto j = 0; j != ms3.extent(1); ++j) {
        for (auto k = 0; k != ms3.extent(2); ++k)
            cout << " " << ms3[i, j, k]; // read via multidimensional index
        cout << '\n';
    }
}
// Output: Level 0: 0 1, 2 3, 4 5, Level 1: 100 101, 102 103, 104 105
}

```

Listing 24.29 This is how the “*mdspan*” added in C++23 works.

You provide the *mdspan* with the raw data, here via `v.data()`. You also set the dimensions when creating it. With `extent()`, you get the size in one dimension. You access the elements of a 3D *mdspan* with `ms3[i, j, k]`.

Previously, you mostly worked with C-arrays of C-arrays and then wrote `mx[y][x]`. The problem with that was that the layout of the data was confusing. You should ideally iterate over the last dimension in the innermost loop to avoid cache misses. In addition, boundary checking is difficult. Overall, *mdspan* greatly simplifies working with multidimensional data. Perhaps *valarray* will soon be obsolete?

The Comma Sequence Operator , Is Deprecated in “operator[]” as of C++20

Did you notice what happened here? You could always write `mx[1, 2, 3]`, but it didn't do what you wanted. The 1 and the 2 were simply ignored because the sequence operator `,` just allows multiple expressions in sequence but only returns the last one as the overall result. So it's `mx[3]`.

Since C++20, you can only use the sequence operator in `operator[]` if you accept a warning, because starting from C++23, the meaning changes! From then on, multidimensional indices are possible.

Inserting into a “vector”

It's not very interesting to put things into a container only during construction; you also want to add things at runtime. The following member functions serve this purpose:

■ Efficient appending at the end

`push_back` and `emplace_back` add a single element to the end of the vector. With `push_back`, the passed value is copied into the vector, while `emplace_back` takes constructor arguments of the new element as parameters and creates it directly in place. On average, appending at the end is optimally fast, in $O(1)$. Starting from C++23, you can also append a range with `append_range`.

■ Insert somewhere

You can insert an element at any position in the vector. With `insert`, you copy it in; with `emplace`, you create it in place with its constructor arguments. The `insert_range` method is available from C++23. Use these methods only in exceptional cases, as vector is slow when inserting at positions other than the end.

With each insertion operation, it may happen that space for the new element has to be created in such a way that all previous elements have to be moved (or copied) to a new (twice as large) location. For this reason, all iterators, spans, and ranges (and pointers; see data) in the container should be considered invalid after *any* insertion operation—unless you have previously ensured with `reserve` or `capacity` that such a reallocation will not occur. As far as time consumption is concerned, you don't need to worry: even if a single insertion operation may sometimes take $O(n)$ time, the algorithm is clever enough to guarantee $O(1)$ on average.

An example for `insert` is shown in [Listing 24.24](#). Because you normally use `push_back`, see [Listing 24.30](#) for a simple example of it.

```
// https://godbolt.org/z/n6En49dsh
#include <vector>
#include <string>
struct President {
    std::string name_; int year_;
    President(std::string name, int year) // name is by Value
        : name_{std::move(name)}, year_{year} // move saves an additional copy here
    { }
};
int main() {
    using namespace std; // enable string literals
    vector<President> presidents{};
    presidents.emplace_back("Washington", 1789);
    presidents.emplace_back("Lincoln", 1861);
    presidents.emplace_back("Kennedy", 1961);
    // ...
    vector<int> v{};
    v.reserve(100);
```

```
    for(int idx = 0; idx < 100; ++idx)
        v.push_back(idx);
    // ...
}
```

Listing 24.30 Typically, you append at the end in vector.

`emplace_back` takes the same arguments as the constructor of the element you want to append. The container ensures that the new object is created in place and does not need to be copied again. Here, for example, it is the string literal "Lincoln"s that benefits from using `emplace_back` and (potentially) needs to be copied one less time than if you wrote the following:

```
presidents.push_back(President{"Heuss"s, 1949});
```

Deleting from a “vector”

There is `erase` as a method for *all* containers except `array`, and in `forward_list` there is `erase_after`. In `vector`, you can either pass an iterator to remove a single element or a pair to remove an entire range:

- **All elements**

`clear()` removes all elements from the vector. Its size is then zero.

- **Last element**

`pop_back()` is the best method to remove a single element from a vector. It is the counterpart to `push_back`. It takes $O(1)$.

- **Single arbitrary element**

`iterator erase(const_iterator where)` deletes a single element. All elements with a larger position than the deleted one are shifted (or copied) one position to the left to fill the gap. This takes $O(n)$ time for a *vector* and should therefore not be done repeatedly.

- **Range**

`iterator erase(const_iterator from, const_iterator to)` deletes all elements between `from` and `to`, exclusive of `to`—mathematically noted as a half-open interval `[from..to)`. For clarification, `erase(begin(), end())` is equivalent to `clear()`. Here too elements shift left to fill the gap. This takes $O(n)$ time for a *vector*.

After the delete operation, the vector is smaller by one element or by the distance between the two iterators.

The return value is an iterator pointing to the first shifted element.

Only Container Methods Actually Delete

As a rule of thumb: Only methods like `erase` actually delete elements from a container, thus changing its size. Free functions like `remove` from `<algorithm>` do *not* delete, but

merely rearrange elements—for example, moving them to the end where they can be easily deleted. This also applies to ranges—that is, everything from the `std::ranges` namespace.

You can delete all elements most effectively with a simple call to `clear()`:

```
std::vector v{ 1, 2, 3 };
v.clear(); // now v is empty
```

You delete a single element with `erase`:

```
// https://godbolt.org/z/6836bdczd
std::vector v{ 1, 2, 3, 4, 5, 6 };
for(auto it=v.begin(); it!=v.end(); ++it) {
    it = v.erase(it);
}
// Here v is: { 2, 4, 6 }
```

The return value of `erase` is an iterator pointing to the first element beyond the erased one. Because the loop increments with `++it` again, this loop deletes every second element.

If you want to delete a range, also use `erase`, but specify the range with two iterators.

```
// https://godbolt.org/z/fe8bev9Y9
std::vector v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin()+2, v.begin()+5); // v is now {1, 2, 6}
v.erase(v.begin(), v.end());      // deletes the rest
```

Listing 24.31 “erase” with two parameters deletes an entire range.

The first parameter is the first element to be deleted; the second parameter is the first element that should remain after the range to be deleted, so it points to an element after the last element to be deleted.

To delete at the end of a vector, `pop_back` is more suitable:

```
std::vector v{ 1, 2, 3 };
v.pop_back(); // v becomes {1, 2}
```

When you combine `erase` with free functions like `remove` or `remove_if` from `<algorithm>`, you have many flexible, elegant, and efficient options to remove elements from a vector—or indeed from any container. You can find some examples in [Chapter 25](#).

Specialty: Size and Capacity

In [Listing 24.32](#), you can see a few operations for the *size* and *capacity* of a vector. Understanding these two metrics is fundamental for the efficient use of vector.

```
// https://godbolt.org/z/YzGK1WazE
#include <vector>
#include <iostream>
#include <format> // C++20
using namespace std;
ostream& operator<<(ostream&os, const vector<int> &vec) {
    for(auto &elem : vec) { os << elem << ' '; } return os;
}
int main() {
    vector<int> data {};           // creates an empty vector
    data.reserve(3);              // Space for 3 elements
    cout << format("{} / {} \n", data.size(), data.capacity()); // 0/3
    data.push_back(111);          // add an element
    data.resize(3);               // Resize; here it appends new elements
    data.push_back(999);          // add another element
    cout << format("{} / {} \n", data.size(), data.capacity()); // 4/6
    cout << data << '\n';         // 111, 0, 0, 999
    data.push_back(333);          // add another element
    cout << data << '\n';         // 111, 0, 0, 999, 333
    data.reserve(1);             // nothing happens, because capacity() > 1
    data.resize(3);              // shrink; elements are removed
    cout << data << '\n';         // 111, 0, 0
    cout << format("{} / {} \n", data.size(), data.capacity()); // 3/6
    data.resize(6, 44);          // resize again, fill with 44s
    cout << data << '\n';         // 111, 0, 0, 44, 44, 44
}
```

Listing 24.32 Operations for the size and capacity of a “vector”.

24.6.4 “array”

An array is an exception among containers in that it is the only container that cannot change its size. You determine at compile time how many elements it contains, and for the duration of its existence, it maintains that size. You cannot add elements or remove elements—only replace existing elements.

Therefore, you also specify the size of this container as a template parameter within angle brackets—for example, `array<int,5>` for an array with five elements. The `array<int,5>` type is thus different from the `array<int,4>` type because that is the nature of templates. Consequently, you cannot write a single function that can handle different array sizes as parameters. These are two overloads:

```
// https://godbolt.org/z/Y1W8WfxYn
#include <array>
void calculate(const std::array<int,4>& data) { /* ... */ }
void calculate(const std::array<int,5>& data) { /* ... */ }
```

If you don't want to write multiple overloads, you need a function template that takes at least the array size as a template parameter.

```
// https://godbolt.org/z/nqbKT9GEa
#include <array>
#include <numeric> // accumulate
#include <iostream>
template<size_t SZ>
int sumSz(const std::array<int,SZ>& data) {
    int result = 0;
    for(auto i=0uz; i<SZ; ++i) // uz is the suffix for size_t since C++23
        result += data[i];
    return result;
}
template<typename Elem, size_t SZ>
Elem sumElem(const std::array<Elem,SZ>& data) {
    Elem result {0};
    for(auto i=0uz; i<SZ; ++i)
        result += data[i];
    return result;
}
// C++20 concept input_iterator, otherwise typename
template<std::input_iterator It>
int sumIt(It begin, It end) {
    return std::accumulate(begin, end, 0);
}
// abbreviated function template with auto
int sumAuto(std::input_iterator auto begin, std::input_iterator auto end) {
    return std::accumulate(begin, end, 0);
}
int main() {
    using namespace std;
    array<int,4> a4 { 1, 2, 5, 8 };
    cout << sumSz<4>(a4) << '\n'; // Output: 16
    array<int,5> a5 { 1, -5, 3, 7, 2 };
    cout << sumElem(a5) << '\n'; // Output: 8
    array<int,6> a6 { 1,2,3, 4,5,6 };
    cout << sumIt(a6.begin(), a6.end()) << '\n'; // Output: 21
}
```

```
array a7 { 1,2,3, 4,5,6, 7 }; // array<int,7>
cout << sumIt(a7.begin(), a7.end()) << '\n'; // Output: 28
cout << sumAuto(a7.begin(), a7.end()) << '\n'; // Output: 28
}
```

Listing 24.33 Different “arrays” as parameters require template programming.

For demonstration, I call `sumSz<4>(a4)` here with the explicit template parameter `<4>`. As you can see with `sumElem(a5)`, it also works without it. And because I don't want to withhold from you that the task of summing elements of a container is already solved in the standard library, `sumIt` uses the corresponding function `accumulate` from `<numeric>`. You always achieve full flexibility regarding containers when you use iterators as template parameters, as shown here. With `array a7`, you can also see that the compiler can deduce both template parameters from the constructor arguments. In `sumIt`, I use `template<std::input_iterator It>`, a C++20 concept, which you can replace with `typename` if you are not yet using C++20. In `sumAuto`, `auto` as a parameter type further shortens the function header to abbreviate the template syntax.

The `array` is thus most similar to the fixed-size C-array. `array<int,100>` corresponds roughly to `int[100]`—with the crucial difference that an `array` does not decay to a pointer and thus retains its size information even after being passed as a function parameter.

I will cover all methods of `array` here as well as the most useful free functions and other tools. Much is similar to `vector`, so I will be brief. You will find a directly transferable detailed discussion of the corresponding feature of `vector` in [Section 24.6.3](#).

All examples in this section for `array` should be considered embedded in the following code.

```
// https://godbolt.org/z/PrM4fGcWo
#include <array>
#include <iostream>
using std::array; using std::cout;
template<typename Elem, size_t SZ>
std::ostream& operator<<(std::ostream&os, const array<Elem,SZ>&data) {
    for(auto &e : data) {
        os << e << ' ';
    }
    return os;
}
int main() {
    // Example code here
}
```

Listing 24.34 Template for the example listings of this section on “array”.

Initialize

The initialization operations for array are:

- **Default constructor**

When you define `array<int,4> x{}`, all elements are zero initialized. You can also initialize all—or some—elements according to the rules of *aggregate initialization* with `array<int,4>{1,2,3,4}` or `array<int,4>{1,2}`.

- **Copy constructor, move constructor**

You can copy arrays of the same size and thus pass and return them by value. Moving is also possible if the element type allows it.

Assignment

You have the following operations available to modify all content of an array at once:

- **Assignment operator**

You can assign new values to an array by assigning it another array with a compatible element type and the same size. `operator=` is therefore defined as expected.

- **Assign**

There is no `assign` because array also does not have a complete set of constructors.

- **Fill**

However, there is `fill`, which allows you to set all values of an array at once, as you can see in [Listing 24.35](#).

Insert and Delete

Inserting and deleting elements is not possible with array.

Read and Write

You can read and write parts of an array in the following ways:

- **Index access**

However, you can access the elements unprotected or protected with `[]` and `at`. You get back a reference that you can read and overwrite.

- **Iterators**

The iterators of array are contiguous and random-access, meaning they can move forward, backward, and be arbitrarily positioned, similar to vector.

- **Beginning and end**

You also have `front` and `back` available, which give you references to the first and last elements of the array, respectively.

- **Tuple interface**

You can read and write an array like a tuple using `std::get<>`, as you can see in [Listing 24.36](#).

■ Span

Since C++20, array can work wonderfully with span, just like vector. From C++23 onward, you can also overlay an *mdspan*.

■ C interface

Because the elements of an array are contiguous like those of a vector, you can use `data()` to get a pointer to the first element and then handle it like a C-array or C-pointer. With slight modifications, [Listing 24.25](#) also works with array.

■ Fill

Specifically for array, there is `fill`, which writes all elements simultaneously.

```
// https://godbolt.org/z/7xY7eqj5Y
array data{ 1,4, 69, 3}; // array<int,4>
data.fill(8);           // now {8, 8, 8, 8}
```

Listing 24.35 “fill” writes all values in an “array” at once.

Specialties: Tuple Interface

From an array, you can not only access elements with the usual container methods, but you can also use an array like a tuple. A detailed description of tuple follows in [Chapter 28, Section 28.1](#). For this, there is an overload of the free function `std::get<size_t>` (`std::array<...>`).

```
// https://godbolt.org/z/KrbTrx1aG
array<int,5> data{ 10, 11, 12, 13, 14};
cout << std::get<2>(data) << '\n'; // 12
```

Listing 24.36 “array” supports “get” from “tuple”.

As with tuples, the index must be a `constexpr` as it is a template argument to `get`.

There is more similarity to tuple, as the `tuple_element` and `tuple_size` class templates are also declared for array, in case you need them for your metaprogramming.

One of the most practical things is undoubtedly that you can *bind* the elements of an array like a tuple *structured* to multiple declaring variables.

```
// https://godbolt.org/z/h6nrPYWfM
#include <array>
#include <iostream>
int main() {
    std::array ports { 80, 443 };           // array<int>
    auto [ http, https ] = ports;          // declares variables and binds them
    std::cout << http << " " << https << "\n";
    auto const& [ rhttp, rhttps ] = ports; // Reference and const are possible
    std::cout << rhttp << " " << rhttps << "\n";
}
```

Listing 24.37 An array supports structured binding.

Here, `int http` and `int https` are defined and initialized with the values 80 and 443. For `rhttp` and `rhttps`, you can see that references to the array also work—here, `const int&`.

Specialties: Comparisons

You can compare two arrays of the same type with all comparison operators: `<`, `<=`, `>`, and so on. You get the *lexicographical* result. For arrays, `operator<=` is defined, from which all other comparisons are derived. The operator behaves as if the free function `lexicographical_compare_three_way()` from the `<algorithm>` header is called. There, `<=` or `<` is used, depending on the availability for the element type. Instead of `operator<=`, all comparison operators for containers were overloaded until C++20. In both cases, this means that the first elements of both arrays are compared first, and if they are not equal, the result is determined. If they are equal, the second element is compared in the same way, and so on. If all elements are equal, then the two arrays are also equal.

```
// https://godbolt.org/z/j6EarszMT
// intentionally array& as arguments
void all(const array<int,4> &a, const array<int,4> &b) {
    cout << "{"<<a<<"}" compared with {"<<b<<"} is "
        << (a < b ? "<, " : "")
        << (a <= b ? "<=, " : "")
        << (a > b ? ">, " : "")
        << (a >= b ? ">=, " : "")
        << (a == b ? "==, " : "")
        << (a != b ? "!=", " : "")
        << '\n';
}

int main() {
    array a{10,10,10,10};
    array b{20, 5, 5, 5};
    array c{10,10,5,21};
    array d{10,10,10,10};
    cout << (a < b ? "smaller\n" : "not smaller\n"); // "smaller", because 10 < 20
    cout << (a < c ? "smaller\n" : "not smaller\n"); // "not smaller", because
                                                    // not 10 < 5
    cout << (a == d ? "equal\n" : "not equal\n");    // "equal", because all are 10
    for(auto &x : {a,b,c}) {
        for(auto &y : {a,b,c}) {
            all(x,y);
        }
    }
}
```

Listing 24.38 Arrays can be compared lexicographically.

The two `for` loops at the end compare the arrays `a`, `b`, and `c` pairwise by calling `compare` for every two arrays. Then `compare` writes the result as text for all comparison operators, so you get this output (excerpted here):

```
{10 10 10 10 } compared with {10 10 10 10 } is <=, >=, ==,  
{10 10 10 10 } compared with {20 5 5 5 } is <, <=, !=,  
{10 10 10 10 } compared with {10 10 5 21 } is >, >=, !=,  
{20 5 5 5 } compared with {10 10 10 10 } is >, >=, !=,  
{20 5 5 5 } compared with {20 5 5 5 } is <=, >=, ==,  
{20 5 5 5 } compared with {10 10 5 21 } is >, >=, !=,  
{10 10 5 21 } compared with {10 10 10 10 } is <, <=, !=,  
{10 10 5 21 } compared with {20 5 5 5 } is <, <=, !=,  
{10 10 5 21 } compared to {10 10 5 21 } is <=, >=, ==,
```

24.6.5 “deque”

The *double-ended queue* is very similar to the vector. It forgoes the guarantee of placing elements consecutively, and thus it offers not `data`, `capacity`, nor `reserve`. However, in addition to the vector, it can efficiently add and remove elements at the *front* in $O(1)$ time: using `push_front`, `emplace_front`, `pop_front`, and from C++23 also `append_range` and `prepend_range`.

One of the most obvious things you can do with a deque—and the reason for its name—is to add things at one end while removing things at the other end. This is the *first in, first out* (FIFO) principle.

```
// https://godbolt.org/z/Ee5anM9as  
#include <deque>  
#include <iostream>  
#include <string>  
#include <cctype>    // toupper  
#include <iomanip>    // boolalpha  
using namespace std;  
bool isPalindrome(string_view word) {  
    // check from both ends simultaneously  
    deque<char> deq{};  
    for(char ch : word) {  
        deq.push_back(toupper(ch));    // uppercase  
    }  
    auto ok = true;  
    while(deq.size() > 1 && ok) {  
        if(deq.front() != deq.back()) {  
            ok = false;  
        }  
    }  
}
```

```

    deq.pop_front();           // Hello deque!
    deq.pop_back();
}
return ok;
}
int main() {
    cout << boolalpha; // Print 'true' and 'false' instead of '1' and '0'
    cout << isPalindrome("Abrakadabra") << '\n';    // false
    cout << isPalindrome("Kajak") << '\n';          // true
    cout << isPalindrome("EvilOlive") << '\n';        // true
    cout << isPalindrome("Aibohphobia") << '\n';      // true
    cout << isPalindrome("Madam") << '\n';            // true
    cout << isPalindrome("") << '\n';                 // true
}

```

Listing 24.39 We remove pairs from the front and back and compare.

Palindromes are words that read the same forwards and backwards. This program checks words for this property. In a loop, `isPalindrome` checks if the first and last letters are the same, continuing until it finds a difference. Then it removes a letter from both ends, and that's where `pop_front` comes into play, which is only available in `deque`.

Yes, with an index loop we could have read and compared directly in `word`, but this way I could demonstrate the `deque` compactly. With C++20 ranges, it is even elegant:

```

namespace rs = std::ranges; namespace vs = std::ranges::views;
bool isPalindrome(const string_view word) {
    return rs::equal(word, word | vs::reverse, rs::equal_to{}, ::toupper,
        ::toupper);
}

```

`equal` is a range algorithm, and `reverse` is a view. `equal_to` is the default anyway, but the two `to_upper` *projections* are applied to the elements of the ranges before comparison.

Typical applications for `deque` include the following:

- A *scheduler* that adds new tasks to the back of the queue and removes completed ones from the front.
- An *undo history* of limited length, where executed actions are inserted at the front, removed from there when they need to be undone, and elements are dropped at the back to limit the depth of the history.
- Text search with pattern matching using regular expressions or their implementation through the simulation of a nondeterministic finite automaton. The nondeterministic transitions are placed in the double-ended queue for processing.

I will cover all methods of `deque` as well as the most useful free functions and other tools. Much is similar to `vector`, and I will be brief in those cases. You will find a directly transferable detailed discussion of the corresponding feature of `vector` in [Section 24.6.3](#).

`deque` does not provide an interface for its capacity like `vector` does. Therefore, the `reserve` and `capacity` methods are not available as they are in `vector`.

Because `deque` is otherwise very similar to `vector`, refer to the examples there. To save you from flipping through pages, I have included some crucial operations of `deque` in [Listing 24.40](#).

```
// https://godbolt.org/z/d3ExcWP4a
#include <deque>
#include <iostream>
#include <iterator> // ostream_iterator
#include <algorithm> // copy
using namespace std;
ostream& operator<<=(ostream& os, int val) { return os<<val<<'\n'; }
int main() {
    deque d{ 4, 6 };           // Create deque<int> with elements
    // Insertion
    d.push_front(3);           // at the front: efficient
    d.insert(d.begin()+2, 5);   // in the middle: slow
    d.push_back(7);             // at the end: efficient
    // Access
    cout <<= d.front();         // from the front: efficient
    cout <<= d[3];               // in the middle: efficient
    cout <<= d.back();           // from the end: efficient
    // Size
    cout <<= d.size();           // read size
    d.resize(4);                // resize (or extend)
    // Iterators
    copy(d.begin(), d.end(), ostream_iterator<int>(cout, " "));
    cout << '\n';               // Output: 3 4 5 6
    // Remove
    d.pop_front();              // at the front: efficient
    d.erase(d.begin()+1);       // in the middle: slow
    d.pop_back();               // at the end: efficient
    d.clear();                  // clear
}
```

Listing 24.40 What the “deque” can do!

Note my comment on `<<=` in the box that followed [Listing 24.12](#).

Initialize

You initialize a deque just like a vector: using the default constructor without arguments, an initializer list, with an initial size, a pair of iterators, or another container for copying/moving. Starting from C++23, initialization with *from_range* is also available.

Assignment

The `assign` method, like in `vector`, has the same options as the constructors. And with `operator=`, reassignment by copy and move is possible. With `swap`, you can exchange the contents of two deques in $O(1)$. Starting from C++23, *assign_range* is also available.

Insert

The deque can efficiently insert elements at both the front and back compared to the vector:

- **Efficient appending at both ends**

From `vector`, the deque has `push_back` and `emplace_back` for appending at the back. For the front, there are `push_front` and `emplace_front`, which also run optimally fast in $O(1)$. Starting from C++23, you can also efficiently append ranges with *append_range* and *prepend_range*.

- **Inserting at any position**

There is no difference from `vector` here. Inserting somewhere with `emplace` or `insert` requires copying or moving many elements and thus takes $O(n)$ time. The *insert_range* method is available starting from C++23.

Deleting

Unlike with `vector`, you can also efficiently remove elements from the front with the deque:

- **All elements**

`clear()` removes all elements from the deque.

- **Last or first element**

`pop_back()` works for deque like it does for `vector`; remove from the front with `pop_front()` in $O(1)$.

- **Single arbitrary element**

iterator `erase(const_iterator where)` deletes a single element like in `vector` in $O(n)$.

- **Range**

iterator `erase(const_iterator from, const_iterator to)` deletes all elements between `from` and `to` like in `vector` in $O(n)$.

24.6.6 “list”

With `list`, things get more interesting. While many familiar methods are again present, which I have already introduced to you with `vector`, there are interesting differences. For one example or another—especially related to iterators—you can still refer to [Section 24.6.3](#), but you will see the specialties of the `list` here.

`list` stores the elements in a doubly linked list. This means that each element is located independently of all other elements somewhere in memory. The elements are not allocated all in one go, but individually. This is an advantage if you have, for example, large elements, because then space is not reserved for many at once. However, with many small elements, it can be a disadvantage because a single memory request costs the system memory and time.

Each element is accompanied by two pointers, one pointing to the predecessor and the other to the successor. This is extra memory that a `vector`, for example, does not need, but it allows you to insert an element anywhere in a `list` by simply “hooking” it between two elements; no shifting actions like in `vector` are necessary. And even when appending at the end or beginning, the entire container content does not have to be occasionally moved to a new, larger space.

The linked pointers are the only way to navigate through the `list`. Direct addressing via an index is not possible. You get *bidirectional iterators* with `begin()` and `end()` and then navigate through the elements with `++` and `--`. Therefore, the ranges generated from a `list` are also only *bidirectional* and not randomly accessible.

Initialize

`list` has exactly the same constructors defined as `vector` and `deque`. This means you have the default constructor without arguments available for an empty list and complete copying or moving, as well as variants for initializing with a number of identical copies of a value, the initializer list, iterator pairs from another container, and, from C++23 onward, the *from_range* constructor. `swap` efficiently swaps entire lists.

Assignment

`operator=` can both copy and move the contents of another `list`. `assign` for reinitialization is overloaded for a number of identical copies of an element, an initializer list, and for a pair of iterators from another container. *assign_range* is available from C++23.

Accessing

There is no index-based access. It is best to work with iterators. Direct element access only works at the ends of the list:

- **Direct element access**

`front()` and `back()` provide you with a reference to the first and last element of the list, respectively. Make sure the list is not empty; otherwise, this access is invalid.

- **Iterator access**

`begin()` and `cbegin()` return a bidirectional iterator `it` to the first element, `end()` and `cend()` point to the end of the list—that is, *behind* the last element. As usual, increment and decrement it with `++it` and `--it`. You can also use `next(it)` and `prev(it)` from the `<iterator>` header.

- **Iterate backward**

Likewise, not uncommon are the `r...` variants of the iterator methods. For example, with `rbegin()` you get an iterator `it` that points to the *last* element. If you increment it with `++` once (yes, *increment*, not *decrement*), it then points to the second to last, and so on.

Note that for `list`, operations that move an iterator `it` by several positions `n` are costly, meaning they take $O(n)$ time. These operations are `prev(it, n)`, `next(it, n)`, and `advance(it, n)`, as well as `distance(it1, it2)` when the iterators have a distance of `n` between them.

For this—unlike with `vector` and other contiguous counterparts—iterators remain valid when you delete or insert an element from the container.

Insert

All insertion operations of `list` are optimally efficient with $O(1)$ per new element. This is particularly true compared to `vector` for insertion at the front and in the middle:

- **Inserting at the end**

You can efficiently add elements to a list at the back with `push_back`, `emplace_back`, and `append_range`.

- **Inserting at the beginning**

This also applies to the front with `push_front`, `emplace_front`, and `prepend_range`.

- **Inserting at any position**

`insert` can insert a single element or an entire area anywhere if you provide a pair of source iterators in addition to the insertion position. Starting from C++23, there is also `insert_range`. `emplace` inserts a single element at the specified position without copying by forwarding the call parameters to an element constructor.

Insert operations do not invalidate other iterators that refer to the list.

Deleting

You can delete all elements with `clear()` and delete one or some with `erase`, just like with `vector`.

Deletion in `list` is optimally fast with $O(1)$ per deleted element. This also applies to deletion at the front or at any arbitrary position.

Iterators that refer to other positions in the container remain valid when elements are deleted.

Specialty: List Operations

You do not sort a list with `std::sort` from `<algorithm>`, as it would require random-access iterators. Here you use the `sort` method of `list`.

The same applies to some other algorithms that do not work with `list` using free functions because when they want to remove elements, they want to move them to the back and swap them randomly. These are `remove`, `remove_if`, and `unique`. `reverse` and `merge` also exist as list methods for reasons of random access. The special implementations in the form of methods are just as efficient as their counterparts from `<algorithm>`. The disadvantage, however, is that these methods always operate on the entire list as they do not receive parameters as range markers in the form of iterators.

A special treat, however, is the `splice` method: the nature of a `list` compared to all other containers—with the exception of the `forward_list`—allows two arbitrarily long lists to merge into one in no time! No matter how many elements are in lists `a` and `b`, after `a.splice(a.end(), b)`, all elements from `b` are in `a`, and this only takes $O(1)$ time.

```
// https://godbolt.org/z/so5Geb4M
#include <list>
#include <iostream>
using std::list; using std::cout; using std::ostream;

ostream& operator<<=(ostream&os, const list<int> &data)
{ for(auto &e:data) os<<e<<' '; return os<<'\n'; }

int main() {
    list numa { 1, 3, 5, 7, 9 };
    list numb { 2, 4, 6, 8 };
    auto wh = numa.end();
    numa.splice(wh, numb); // transfer in O(1)
    cout <<= numa; // Output: 1 3 5 7 9 2 4 6 8
    cout <<= numb; // Output: (none)
}
```

```

    numa.sort();    // sort as a method, not from <algorithm>
    cout <<= numa; // Output: 1 2 3 4 5 6 7 8 9
}

```

Listing 24.41 “splice” is the specialty of “list” and efficiently connects two lists.

Note my comment on `<<=` in the box that followed [Listing 24.12](#).

With the other overloads of `splice`, you can transfer a single element or a range of n elements. Then the efficiency class is $O(n)$. One often-overlooked methods is `merge`. With this, you can combine two *sorted* lists into one sorted large list. This is better than combining two lists with `splice` followed by `sort`.

24.6.7 “forward_list”

The `forward_list` is different from all other containers. It stores its elements in a singly linked list. As a result, it has the smallest possible memory overhead for a list—namely, just one pointer per stored element.

However, this means that you can only navigate forward from an element and not backward. For example, if you want to insert an element at a given position `it`, you cannot reach the element before it to adjust its linking pointer.

As a solution, the designers of the standard library decided not to offer the necessary methods like `insert`, `emplace`, `erase`, and `insert_range`, as these could not be implemented efficiently. Instead, there are `insert_after`, `emplace_after`, `erase_after`, and `insert_range_after`, which act on an element “later” than is usual with other containers. For everyday use, this means that `forward_list` is operated completely differently from all other containers.

I therefore recommend using `forward_list` only when you *really* need it and all other containers have been ruled out as options.

Many of the methods are still identical to those of `vector` or `list`, especially those that relate to the entire container. I will therefore be brief here and only go into detail about the specialties of `forward_list`. For more examples, refer to [Section 24.6.3](#).

Initialize

The `forward_list` offers all the usual container constructors that `list` and `vector` also offer: by default, with an initial size, a number of identical element copies, a complete copy or move, the initializer list, or a pair of iterators from another container. Starting from C++23, there is also a constructor with the `from_range` marker.

Assignment

`operator=` can copy and move like in `list` and `vector`. `assign` can assign an initializer list, a pair of iterators from another container, or a single value in multiple copies. From C++23, there is also `assign_range`. `swap` exchanges the whole content of two lists.

Accessing

As with `list`, there is no index-based access. You also cannot navigate backward through a `forward_list`:

- **Direct element access**

`front()` gives you a reference to the first element of the list. The list must not be empty.

- **Iterator access**

`[c]begin()` gives you a forward iterator `it` to the first element, and `[c]end()` points past the last element. As usual, increment it with `++it`. Ranges generated from a `forward_list` are *forward ranges* and thus suitable for many, but by no means all, algorithms. For `forward_list`, there are the `before_begin()` and `cbefore_begin()` methods, which give you a “virtual” iterator one element *before* `begin()`. The returned iterator has a special role and can be used in conjunction with the `...after` methods. You can see an example in [Listing 24.42](#).

The iterator returned by `before_begin()` and `cbefore_begin()` must not be dereferenced. However, you need it to insert an element at the very beginning of the list using `insert_after`, for example. If you increment it with `++it`, it is identical to `begin()` or `cbegin()`.

Iterators remain valid when you delete or insert an element in the `forward_list`.

Useful Iterator Traits

In the `<iterator>` header, you will find useful tools related to iterators. For example, you can get a copy of `it` advanced by one with `next(it)`. I find this particularly useful with `forward_list` because you cannot go back with `--it` or use `it+1`. Thus, you can effectively write `it+1` with `next(it)` without the iterator needing to support `+` as an operation.

Insert

Unfortunately, inserting is rather complicated with `forward_list`. You cannot insert at the position pointed to by an iterator, but only one element behind it. [Figure 24.7](#) and [Figure 24.8](#) show that the `Fili` node is not reachable from `it`. However, this would be necessary to break its link to `Oin` and instead link to the new `Bilbo` node. We can only reach the `Fili` node by means of a loop that starts at `begin()`—and that takes $O(n)$ steps,

so it is out of the question. The alternative remains to bite the bullet and accept a slightly modified interface compared to the other containers.

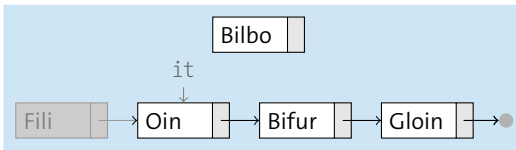


Figure 24.7 From “it”, previous nodes are not accessible.

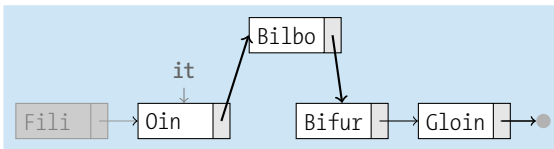


Figure 24.8 To insert, you can only change the linkage of the “it” node.

So, you can basically insert at any position, but you must pay attention to the correct use of the `...after` methods. At any position? No! One single position remains and offers fierce resistance. With the `...after` methods, you actually cannot insert *at the very beginning*. To do so, you have two options: either you use the method `push_front` inherited from `list`, or you get the special `before_begin()` iterator. It points to a non-existent element before `begin()`, which serves the specific purpose of being used as an argument for methods such as `insert_after` and the like.

```
// https://godbolt.org/z/9b9fY4Mve
#include <iostream>
#include <forward_list>
int main()    {
    std::forward_list mylist = {20, 30, 40, 50};
    mylist.insert_after(mylist.before_begin(), 11 );
    for (int& x: mylist) std::cout << ' ' << x; //Output: 11 20 30 40 50
    std::cout << '\n';
}
```

Listing 24.42 You can use “before_begin” as an argument for “insert_after”.

Have I already mentioned that you should only use `forward_list` if you really need it?

These are the options for inserting into a `forward_list`:

■ Inserting at the beginning

With `push_front` and `emplace_front`, you can insert a new element at the beginning. Alternatively, you can use `con.insert_after(con.before_begin(), ...)`. From C++23, there is also `prepend_range`.

■ Inserting at any position

`insert_after` and `emplace_after` insert element(s) somewhere using a position iterator. `insert_after` can handle a single element or multiple elements from an initializer list or a pair of iterators. If you want to insert at the very beginning, you need the special iterator from `before_begin`, as you can see in [Listing 24.42](#). `insert_range_after` is available from C++23.

Like `list`, all insertion operations of `forward_list` are optimally efficient with $O(1)$ per new element, though cumbersome.

Insert operations do not invalidate other iterators that refer to the list.

Deleting

You can delete all elements as with any other container using `clear()`.

If you want to delete at a specific position, you have to consider the same things as when inserting: you either use `pop_front()` or one of the `..._after` methods with the same complications and possibly the help of the special `before_begin()` method.

■ Delete at the beginning

With `pop_front`, you delete the first element. Alternatively, you can also use `con.erase_after(con.before_begin())`.

■ Delete somewhere

`erase_after` deletes a single element directly after the given position or multiple elements if you pass a pair of iterators.

```
// https://godbolt.org/z/andfoh18z
#include <forward_list>
#include <iostream>
#include <iterator> // next
using std::cout; using std::forward_list; using std::ostream;
ostream& operator<<=(ostream&os, const forward_list<int> &data)
    { for(auto &e:data) os<<e<<' '; return os<<'\n'; }

int main()    {
    forward_list nums {40, 50, 60, 70};
    cout <<= nums;                // Output: 40 50 60 70
    nums.insert_after(nums.before_begin(), {10, 20, 30});
    cout <<= nums;                // Output: 10 20 30 40 50 60 70
    auto wh = std::next(nums.begin(), 2); // two elements forward
    auto to = std::next(wh, 3);           // three elements after where
    nums.erase_after(wh, to);
    cout <<= nums;                // Output: 10 20 30 60 70
}
```

Listing 24.43 “`erase_after`” can delete a range of elements.

Like `list`, `forward_list` deletes each individual element in $O(1)$ optimal speed.

Note my comment on `<<=` in the box that followed [Listing 24.12](#).

Iterators to other positions remain valid when deleting.

Specialty: List Operations

Just like `list`, `forward_list` has a set of methods that serve as replacements for some free functions from `<algorithm>`, but always operate on the entire list instead of just a range between two iterators. Fortunately, `merge` is also available for merging two pre-sorted lists.

The list specialty of efficiently merging entire lists or list ranges works again with a special case in `forward_list` compared to `list`—namely, `splice_after`, as shown in [Listing 24.44](#).

```
// https://godbolt.org/z/Ka47W5Wjh
#include <forward_list>
#include <iostream>
using std::cout; using std::forward_list; using std::ostream;

ostream& operator<<=(ostream&os, const forward_list<int> &data)
{ for(auto &e:data) os<<e<<' '; return os<<'\n'; }

int main() {
{
    forward_list fw1 {10, 20, 30, 40};
    forward_list fw2 {5, 6, 7, 8};
    fw1.splice_after(fw1.begin(), fw2); // transfers everything
    cout <<= fw1;                      // Output: 10 5 6 7 8 20 30 40
    cout <<= fw2;                      // Output:
}
{
    forward_list fw1 {10, 20, 30, 40};
    forward_list fw2 {5, 6, 7, 8};
    // one element left:
    fw1.splice_after(fw1.begin(), fw2, fw2.begin(), fw2.end());
    cout <<= fw1;                      // Output: 10 6 7 8 20 30 40
    cout <<= fw2;                      // Output: 5
}
}
```

Listing 24.44 “`splice_after`” can very efficiently merge lists.

`splice_after` comes in several variants to merge entire `forward_list` containers or just ranges of them. Note that the special nature of the `forward_list` interface is once again

a bit tricky. As you can see in [Listing 24.44](#), the two very similar calls to `splice_after` yield different results: The first variant only takes the source list as a parameter and transfers the entire content of the source list. However, if you pass two iterators as the source area, the `forward_list` is not spliced *at* the first iterator but *behind* it; that is, the “after” rule applies again. Because I pass `begin()` here, the splice happens one element after the begin. As a result, one element remains in the source.

24.7 Associative and Ordered

In this section, `Cont` or `C` stands for one of the container types like `set<int>` or `map<int, string>`, and `cont` is for an instance of a container, such as `set<int> cont;`. For *key type*, I use `Key` or `K`, for *target type* `Target` or `T`, and the *value type* `pair<K, T>` is `Value` or `V`. *Iterator types* are abbreviated as `It`.

■ `set<Key>`

This “set” contains elements where the stored values are also the keys, in contrast to the `map`. For example, simply put all `Person` elements into a `set<Person>`. Because the keys are always well-ordered, the elements are now always sorted, allowing fast retrieval in guaranteed time.

■ `map<Key, T>`

Use `map` to translate values of one type to another. `map` finds a corresponding target value based on a key. For example, `map` translates `string` into your own type `Person`. In this container, the `Key` elements are always well-ordered, and the search is guaranteed to be fast.

■ `multimap<Key, T>` and `multiset<Key>`

For each `Key`, the non-`multi...` variants can only store one element. With these variants, multiple elements can also be hidden behind a `Key`. Therefore, duplicate insertion is possible.

Container	Description
<code>set</code>	Sorted and duplicate-free collection of elements
<code>map</code>	Unique keys refer to a target value
<code>multiset</code>	Sorted collection of elements that can occur multiple times
<code>multimap</code>	Sorted keys can each occur multiple times

Table 24.6 Profile: The associative ordered containers.

Starting from C++23, there is also the family of *flat associative ordered container adapters*. What a cumbersome name; I prefer to call them *flat containers*. Especially if you expect many small containers, take a look at `flat_set` and `flat_map`; see [Section 24.9](#).

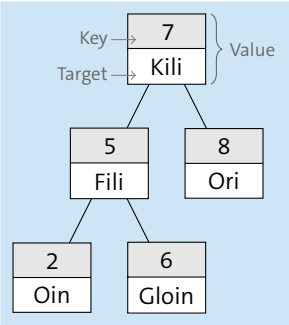


Figure 24.9 A “map” stores values ordered by keys in a tree.

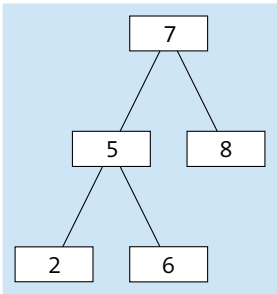


Figure 24.10 A “set” only holds ordered keys.

24.7.1 Commonalities and Differences

In Table 24.7, you can see the main properties that differentiate the ordered associative containers from each other. By *guaranteed*, I mean that the operations are guaranteed to be fast—specifically, $O(\log n)$. The following notes apply to the * entries:

- **Key type**
The ordered associative containers can use almost any type as a key, as long as its values can be ordered with less than $<$. You must be able to create, remove, and copy them.
- **Insert, remove**
Storing the elements in a sorted tree guarantees that you will always find a key in $O(\log n)$ comparison operations. It does not matter in which order you insert the keys or if they are all the same.

Property	set	map	multiset	multimap
Unique keys	Yes	—	Yes	—
Keys to target values	—	Yes	—	Yes
Key type*	$<$	$<$	$<$	$<$
Dynamic size	Yes	Yes	Yes	Yes
Overhead per element	Yes	Yes	Yes	Yes
Memory layout	Tree	Tree	Tree	Tree
Insert, remove*	Guaranteed	Guaranteed	Guaranteed	Guaranteed
Search	Guaranteed	Guaranteed	Guaranteed	Guaranteed

Table 24.7 Properties of ordered associative containers. The meaning of the asterisks is explained ahead.

Property	set	map	multiset	multimap
Sorted	Always	Always	Always	Always
Iterators/ranges	Random	Random	Random	Random
Algorithms	All	All	All	All

Table 24.7 Properties of ordered associative containers. The meaning of the asterisks is explained ahead. (Cont.)

24.7.2 Methods of Ordered Associative Containers

I am not always very precise or detailed with the function signatures here to save space and increase clarity. If I do not include the parentheses with `method`, there are usually multiple overloads:

- **Constructors:** `C()`, `C(Compare)`, `C{...}`, `C(It, It)`, `C(from_range, rg)`
The default constructor `Cont()` creates an empty container. With a pair of iterators from another compatible container, you can copy its contents. You can specify an initializer list that populates your container. For a `map`, this means you specify nested `{...}` because each element must initialize a pair—for example, `map<int, string>{ {2, "two"s}, {3, "three"s} }`. Starting from C++23, there is a `from_range` constructor.
- **Copy and move:** `C(const C&)`, `C(C&&)`, `C& op=(const C&)`, `C& op=(C&&)`, `swap`
You can pass a different `Cont` as an argument to the copy and move constructors and operators. Efficient `swap(Cont& other)` is implemented by all associative containers. *Note:* Compared to sequence containers, associative containers have neither `assign` nor `assign_range`.
- **Destructor:** `~Cont()`
All containers have destructors that remove the contained elements. Note that this does not apply if you pack raw pointers into containers.
- **Iterate forward:** `begin()`, `end()`; and **backward:** `rbegin()`, `rend()`
You can iterate here just like with sequential containers. The ordered containers guarantee that the elements are sorted when iterating. The iterators of the `map` variants reference a `pair<K, T>`.
- **Element access:** `T& operator[] (Key)`, `T& at (Key)`
Only `map` has these methods. If the key does not exist in `at`, an `out_of_range` exception is thrown. With `operator[] (Key)`, you can read and write in the `map`. It does not fail if a key is not found. For example, if you write `auto v = mymap[key]`, a new (default-constructed) target value will be created in `mymap[key]` if necessary. Only then will it be returned to `v`. Handy—and important to know.
- **Size:** `size()`, `empty()`, `max_size()`
With `size()`, you get the current number of elements in the container; `empty()` is

true if this number is zero. With `max_size()`, you get the library-specific maximum size of the container.

- **Modification: `clear()`, `erase`, `insert`, `insert_or_assign`, `emplace`, and `insert_range`**
These methods work like those in sequence containers. In maps, an element consists of a `pair<K,T>`, which you can insert with `insert` and `emplace`. Only map also has `insert_or_assign`. `insert_range` is available from C++23.
- **Search: `count`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `contains`**
With these methods, you can get information about the keys in the container. `count` returns how often one occurs; `find` returns an iterator to a found key (possibly one of several) or `end()` if it is not found. `lower_bound` and `upper_bound` do something similar, but they search for the first and last found key, respectively. If nothing is found, they return an iterator to the position where the searched key would be inserted, unlike `find`. `equal_range` performs both operations simultaneously and returns a pair of iterators that encompass all keys matching the searched one (this has nothing to do with the ranges library). In the non-`multi_...` variants, it is logical that there is at most one element in between, while in the `multi_...` variants, there can be several. `contains` is available from C++20.
- **Transfer: `merge`**
With `merge`, you transfer elements from one container to another. In the `multi_...` variants, all elements are transferred, while in the non-`multi_...` variants, only those not present in the target are transferred, with duplicates remaining in the source.

All ordered associative containers store their elements in a tree structure (see [Figure 24.9](#) and [Figure 24.10](#)). To ensure the efficiency of the container, it is kept *balanced* at all times. This means that with each insert or delete operation, the nodes are cleverly rearranged so that all leaves of the tree are pulled to (almost) the same height. This additional management effort is worthwhile because it *guarantees* that any search will find the element or confirm its absence in at most $\log n$ steps. This holds true at all times, despite constant insertions or deletions, regardless of the data you put in.

24.7.3 “set”

For a `set`, the following main use cases exist:

- **Deduplication**
You want to store a collection of objects, and duplicates should be ignored. Because a `set` stores each key only once, all duplicates are already removed when reading.
- **Sorting**
A collection of objects should be kept sorted when you alternately store, search, store, search, and so on. A `vector` is often more suitable if you first store a collection of objects and then want to search quickly in a later separate phase. A `set` is recommended if there are not two clearly separated phases.

If you do not need this guarantee of constant order, because you, for example, access the `set` in two separate phases—a filling phase and a reading phase—then it may be worthwhile to use a `vector` instead. For this, see an example in [Chapter 26, Section 26.2.1](#). From C++23, there is also the `flat_set` adapter to a `vector`.

The examples in this section for `set` should be thought of as embedded in the code from [Listing 24.45](#).

```
// https://godbolt.org/z/7b8GaE9xT
#include <set>
#include <iostream>
using std::set; using std::cout;
template<typename Elem, typename Comp>
std::ostream& operator<<=(std::ostream&os, const set<Elem,Comp>&data) {
    for(auto &e : data) {
        os << e << ' ';
    }
    return os << '\n'; // '<=<' instead of '<<' for line break
}
int main() {
    // Example code here
}
```

Listing 24.45 This is the template for the example listings in this section on “`set`”.

Note that I overload the `<=<` for operator output with line breaks only to keep the examples compact, as explained in more detail in the box under [Listing 24.12](#).

The Elements and Their Properties

The elements that are packed into a `set` could also be called *keys* because the internal tree structure is built based on them so that they can be quickly found again.

Therefore, the elements must be *sortable*. What does that mean specifically? Two elements `e` and `f` must support `e < f`. That's enough.

...Almost. Because this less-than must follow the usual mathematical rules; otherwise, `set` won't work. That means:

- If `e < f` holds, then `e >= f` must be false (without having to implement `>=`, mind you).
- Therefore it follows that if both `e < f` is false and `f < e` is false, then `e == f` must be true.

For example, if the element type is `int` and `e` and `f` are 5 and 11, then `5 < 11`. It follows that `!(5 >= 11)`, and that is true. If `e` and `f` are both 7, then `e < f` and `f < e`, so `7 < 7` is false, thus `e == f`, and that is also true.

However, if you engage in any nonmathematical folly, it may come to pass that `set` does not function as intended. For example, if you define `<` such that it accidentally defines `<=` on your element type, then `set` can no longer determine equality. If `e` with value 7 is already contained in the set and you search with an `f` also with value 7, then it no longer works.

In [Listing 24.46](#), I defined a lambda as a comparison function so that my example also works on normal `int`, for which there is already a `<`. If you want to put your own data type `Elem` into your set, it is sufficient that you overload

```
bool operator<(const Elem &e, const Elem &f).
```

However, it should be easier to overload `operator<=`, at least from C++20 onward. Because if you do that, C++ automatically derives all comparison operators from it.

[Listing 24.46](#) incorrectly defines the comparison operation for `set` by returning `true` instead of `false` when `e` and `f` are equal. Then, when searching for the element actually contained in the set with `find(7)`, it will no longer be found.

```
// https://godbolt.org/z/jfdh3ddsz
using std::cout; using std::ostream; using std::set;
auto comp = [](auto e, auto f) { return e <= f; }; // ✗ defined wrong!
std::set<int,decltype(comp)> data(comp);
data.insert({ 9,5,7,2,3,6,8 });
cout <<= data; // Output with luck: 2 3 5 6 7 8
auto it = data.find(7);
if(it != data.end()) {
    cout << "got it: " << *it << '\n';
} else {
    cout << "it's gone!" << '\n'; //you will probably end up here
}
```

Listing 24.46 If you define the comparison operation incorrectly, then “set” will no longer work.

However, you can group multiple elements into one category. For example, if it is sufficient for you that the tens place matches to consider equality, your comparison operation can do that.

```
// https://godbolt.org/z/xdPc51q5s
auto comp = [](auto e, auto f) {return e/10<f/10;}; // grouping is okay
std::set<int,decltype(comp)> data(comp);
data.insert({ 14,23,56,71 });
cout <<= data; // Output: 14 23 56 71
auto it = data.find(29); // 29 now also finds 23
```



```
if(it != data.end()) {
    cout << "got it: " << *it << '\n'; // Output: got it: 23
}
data.insert(79);                      // nothing happens, 71 is already in
cout <= data;                          // Output: 14 23 56 71
```

Listing 24.47 The comparison operation can group elements.

So make sure that your comparison operation, if possible, does the intuitive thing. If you have to program something nonintuitive for a good reason, then comment sufficiently, and keep in mind that `<` must adhere to the rules of the mathematical strict weak ordering.

If you have your own data type, it is best to overload `operator<=` and let the compiler derive the necessary `operator<` from it.

```
// https://godbolt.org/z/W9sK4jnzK
#include <iostream>
#include <set>
#include <string>
struct Dwarf {
    std::string name;
    int height;
    auto operator<=>(const Dwarf& other) const {
        return height <=> other.height;
    }
};
int main() {
    std::set<Dwarf> dwarves {
        {"Thorin", 140}, {"Balin", 136}, {"Kili", 138},
        {"Dwalin", 139}, {"Oin", 135}, {"Gloin", 137},
    };
    for(auto &d: dwarves) {
        std::cout << d.name << ' ';
    } // Output: Oin Balin Kili Gloin Dwalin Thorin
}
```

Listing 24.48 A custom spaceship operator for “set” compatibility.

Here, `<=>` only considers the height of the dwarves when comparing. If you want to consider all elements of the struct, then you can let the compiler generate the necessary comparison cascade with `= default`:

```
auto operator<=>(const Dwarf& other) const = default;
```

Here, the output is then sorted alphabetically.

As you can see, you can write `operator<=` as a normal member function. You do not need to define it as a `friend` or even outside the class.

Initialize

In addition to the element or key type `Key`, `set` also has a *comparator* typename `Compare` as a template parameter (as well as the obligatory *allocator*; see [Section 24.3](#)). If you do not specify one, `std::less<Key>` is the default. This ultimately means that two keys `e` and `f` are compared using less than `<` to keep the set sorted. For built-in types like `int`, this is also the built-in `<`. For your own types, you override the free function `operator<`. If you choose one of these options, you do not need to specify your own comparator.

Only if `<` is not sufficient for you should you define your own comparator. You need to define a kind of less-than `<`, but never less-than-or-equal `<=`; see [Listing 24.46](#). The comparator is something callable, with two `Key` parameters and `bool` as return type, or something implicitly convertible to these. This can be, as usual, a function pointer, a lambda, or a functor.

A functor is practical here because then it is enough to specify its type as a template parameter. Alternatively, since C++17, you can use a lambda or a function pointer as a constructor argument, allowing you to omit the other explicit template parameters.

```
// https://godbolt.org/z/c8a9qPoPP
#include <set>
#include <functional> // function
using std::set; using std::function; using std::initializer_list;
bool fcompTens(int a, int b) { return a%10 < b%10; }
struct Fives {
    bool operator()(int a, int b) const { return a%5 < b%5; }
};

int main() {
    // Functor
    set<int, Fives> ff1;
    ff1.insert(5);
    set ff2({5}, Fives{});
    set ff3(initializer_list<int>({}), Fives{});
    // Lambda
    set<int, function<bool(int,int)>> ll1([](auto a,auto b){return a%3<b%3;});
    ll1.insert(3);
    auto lcomp = [](int a, int b) { return a%3 < b%3; };
    set<int, decltype(lcomp)> ll2(lcomp);
    ll2.insert(3);
    set ll3({3}, lcomp);
```

```
// Function pointer
set<int, bool(*)(int,int)> zz1(&fcompTens);           // C-style
zz1.insert(10);
set<int, function<bool(int,int)>> zz2(&fcompTens); // C++ style
zz2.insert(10);
set<int, decltype(&fcompTens)> zz3(&fcompTens);      // C++ style
zz3.insert(10);
}
```

Listing 24.49 There are various ways to specify a comparator.

The comparator can usually be specified in addition to the elements with which the set is to be initialized. Similar to vector ([Section 24.6.3](#)), there are the following ways to pass the initial elements to the constructor:

- Without arguments (e.g., the default constructor)
- In a range from another container using a pair of iterators
- As another set for copying
- As an initializer list
- From C++23, as a range with the tag *from_range*

```
// https://godbolt.org/z/oc9xxev95
// without arguments
set<int> empty{};
cout <<= empty;           // Output:
// initializer list
set list{ 1,1,2,2,3,3,4,4,5,5 }; // set does not take duplicates
cout <<= list;             // Output: 1 2 3 4 5
// copy
set copy(list);
cout <<= copy;             // Output: 1 2 3 4 5
// pair of iterators
set from_to( std::next(list.begin()), std::prev(list.end()));
cout <<= from_to;          // Output: 2 3 4
// Range
set even(from_range, list | vs::filter([](int i){ return i%2; }));
cout <<= from_to;          // Output: 2 4
```

Listing 24.50 There are again several ways to specify elements when constructing.

Shifts are possible because the move constructor is also defined. For this, as usual, `Elem` must be movable:

```
// https://godbolt.org/z/1dTPEd3nP
set source{1,2,3,4,5};
```

```

cout <<= source;           // Output: 1 2 3 4 5
set target( std::move(source) ); // move instead of copy
cout <<= source;           // Output:
cout <<= target;           // Output: 1 2 3 4 5

```

Assignment

operator=(const set<Elem>&) also is offered by set, so you can copy the contents of another set into a previously created set later. And because there is an overload for operator=(set<Elem>&&), this also works with moving:

```

// https://godbolt.org/z/8eGvWY45o
set source{1,2,3,4,5};
set<int> target{};
set<int> target2{};
cout <<= source;           // Output: 1 2 3 4 5
cout <<= target;           // Output:
cout <<= target2;          // Output:
target = source;           // copy afterward
cout <<= source;           // Output: 1 2 3 4 5
cout <<= target;           // Output: 1 2 3 4 5
target2 = std::move(source); // move
cout <<= source;           // Output:
cout <<= target2;          // Output: 1 2 3 4 5

```

The methods `assign` and `assign_range` do *not* exist for associative containers and thus also *not* for `set`. For sequence containers, this method is used to reinitialize an already existing container, providing similar parameter specification options as with a constructor call.

If you want the possibility of such constructor-like reinitialization for associative containers, you need to proceed differently. This can be done without losing performance. [Listing 24.51](#) shows two possibilities.

```

// https://godbolt.org/z/ozMY86ns8
#include <vector>
// ...
set data{1,2,3,4,5};
std::vector source{10, 20, 30, 40, 50};

// There is no set::assign:
data.assign(source.begin(), source.end()); // ⚡ no set::assign
// So simulate it using a temporary set:
set temp(source.begin(), source.end());    // copy from source ...
data.swap(temp);                           // ... efficiently swap contents
cout <<= data;                             // Output: 10 20 30 40 50

```

```
// ... or by clearing first and then inserting:
data.clear(); // clear ...
data.insert(source.begin(), source.end()); // ... and insert
cout <<= data; // Output: 10 20 30 40 50
```

Listing 24.51 Instead of “assign” you can use the copy-and-swap idiom.

In the first variant, I use a temporary set and apply the *copy-and-swap* idiom. The second option utilizes the fact that `insert` supports as many parameter combinations as the constructor.

Insert

In a set, you always insert using `insert` or `emplace`. `insert` supports all possible parameter combinations, similar to those of the set constructor, and from C++23 also `insert_range`. The potentially faster `emplace_hint` is available alongside `emplace`.

All iterators over a set retain their validity after an insertion operation.

Inserting a Single Element with Automatic Positioning

`insert` can insert a single value by copy. The element is automatically inserted at the correct position in the sorted order. To guarantee this, first, the correct position must be found in $\log n$ steps, and second, the internal tree structure likely needs to be rearranged. The same applies to `emplace`, except that you specify the constructor arguments of the element directly instead of the element itself. If what you want to insert already exists in the set, it will not be overwritten! The return value of `insert` in this case is a `pair<iterator, bool>`: if `bool` is true, then a new element was actually inserted. The iterator points to the insertion position.

I let the compiler define the method operator<= for `Punkt` with = default, which then lexicographically compares the fields `x_` and `y_`. The compiler generates operator< from it. If you don't have C++20 available, define operator< as a friend with `return tie(a.x_, a.y_) < tie(b.x_, b.y_);`.

```
// https://godbolt.org/z/8nj7vfeba
// ...
template<typename IT> ostream& operator<<(ostream& os, const pair<IT, bool> wh)
{ return os << (wh.second ? "yes" : "no"); }
struct Point {
    double x_, y_;
    Point(double x, double y) : x_{x}, y_{y} {}
    auto operator<=(const Point&) const = default;
    friend ostream& operator<<(ostream& os, const Point& a) {
        return os << "(" << a.x_ << ',' << a.y_ << ")";
    }
};
```

```

int main() {
    set data{ 10, 20, 30, 40, 50, 60, 70 };
    auto wh = data.insert(35);           // inserts between 30 and 40
    cout << "new? " << wh << '\n';      // Output: new? yes
    wh = data.insert(40);                // already exists, so does not insert
    cout << "new? " << wh << '\n';      // Output: new? no
    set<Point> points{};
    points.insert( Point{3.50,7.25} ); // temporary value
    points.emplace(1.25, 2.00);         // constructor arguments
    cout <<= points;                    // Output: (1.25,2) (3.5,7.25)
}

```

Listing 24.52 To insert a single element, use “insert” or “emplace”.

One variant of `insert` works together with `extract`. The return type of `extract`, which is only known internally, is a parameter of `insert`. This allows you to efficiently transfer elements between similar containers.

Inserting a Single Element with a Position Hint

You can achieve a slight speedup by providing an iterator that points to the insertion position as an additional parameter for `insert`. For `emplace`, use `emplace_hint` so that the iterator is not confused with a forwarded constructor argument. As a return, you get an iterator to the insertion position. If your position hint is correct, the insertion takes constant time on average—that is, $O(1)$. If the hint is incorrect, the insertion still occurs at the correct position regarding the sorting, but the time consumption falls back to that of a normal insertion.

```

// https://godbolt.org/z/qzaj79bor
set data{ 10, 20, 30, 40, 50, 60, 70 };
set<int> target;
auto hint = target.begin();
for(auto &e : data) {
    hint =                               // Use insertion position in the next round
    target.insert(hint, e); // Hint helps because data is sorted
}

```

Listing 24.53 You can reuse the return value when inserting sorted ranges.

The return value of the `insert` method with a position hint returns an iterator for associative containers as well as for sequence containers. This allows you to write functions that work with both container types.

```

// https://godbolt.org/z/h1G8v86jv
#include <set>
#include <vector>
#include <iostream>

```

```
using std::cout; using std::ostream; using std::set; using std::vector;

template<typename Container>
void insFive(Container& cont, int a, int b, int c, int d, int e) {
    auto it = cont.end();
    for(int x : { a, b, c, d, e }) {
        it = cont.insert(it, x); // works with vector, set etc.
    }
}

int main() {
    vector<int> dataVec{ };
    insFive(dataVec, 9, 2, 2, 0, 4 );
    for(auto e : dataVec) cout <<e<<' ';
    cout << '\n'; // Output: 4 0 2 2 9
    set<int> dataSet{ };
    insFive(dataSet, 9, 4, 2, 2, 0);
    for(auto e : dataSet) cout <<e<<' ';
    cout << '\n'; // Output: 0 2 4 9
}
```

Listing 24.54 Use the same insert for sequential and associative containers.

Inserting Multiple Elements

insert also takes a pair of iterators or an initializer list as parameters to insert multiple elements at once. From C++23, there is also *insert_range*. You cannot provide a position hint in this case. You do not get information about whether and which elements were actually added or already existed. The return type is void.

```
// https://godbolt.org/z/rTvovhr5M
#include <vector>
set data{ 10, 20, 30, };
data.insert( { 40, 50, 60, 70 } ); // Initializer list
std::vector new_vec{ 5, 25, 35, 15, 25, 75, 95 };
data.insert( new_vec.cbegin()+1, new_vec.cend()-1 ); // area
cout <<= data; // Output: 10 15 20 25 30 35 40 50 60 70 75
```

Listing 24.55 You can also insert multiple elements.

You can see here that the 25 from new_vec is not inserted twice.

Accessing

Access via Iterators

The iterators of set are bidirectional; see [Table 24.2](#). Because the *ordered associative containers* are ordered at all times (hence the name), you always iterate over the

elements in a given order. With `[c]begin()/[c]end()`, this order is ascending; with `[c]rbegin()/[c]rend()`, it is descending.

Access via Search

The most important search function is `find`. It takes a search element as a parameter and returns an iterator. If the element is found, it points to the found element; if not, it points to `end()`. `lower_bound` behaves somewhat differently: it finds the first element that is not less than the search element. That means if it is present, it points to it; if not, it points to the next larger element. With `upper_bound`, you get the first element that is greater than the search element. Both combined form `equal_range`: you get a pair of two iterators that mark an area enclosing the search element (so, not a C++20 range). With `count`, you find the number of matching keys, as in `auto r = equal_range(key); auto count = distance(r.first, r.second)`. Naturally, in a set, this is either zero or one. Since C++20, there is also `contains`, which returns a `bool`. All searches are guaranteed to take at most $\log n$ steps.

Note the somewhat asymmetric behavior of `upper_bound`: when you search in the set `s` with elements `{1,2,3}` using `it = s.upper_bound(2)`, it points to 3, because it is the first element greater than 2. In contrast, `it = s.lower_bound(2)` returns an iterator to 2, because it is the last element not less than 2. Together, they are exactly what `equal_range` returns as a pair. This is illustrated in [Listing 24.56](#).

```
// https://godbolt.org/z/MW74f97f7
#include <set>
#include <iostream>
using std::cout; using std::ostream; using std::set;
void search(const set<int>&data, int what, ostream&os) {
    os << what << "? ";
    // contains
    auto inside = data.contains(what); // C++20
    os << "inside:" << (inside ? "yes." : "no."); // check contains
    auto where = data.find(what);
    if(where != data.end()) {
        os << " found:" << *where << " ";
    } else {
        os << " not found. ";
    }
    auto lo = data.lower_bound(what);
    if(lo != data.end()) {
        os << "lo:" << *lo;
    } else {
        os << "lo:-";
    }
    auto up = data.upper_bound(what);
    if(up != data.end()) {
```



```
        os << " up:" << *up;
    } else {
        os << " up:-";
    }
    // [lo,up] is now the same as what equal_range would have returned
    os << " Range:{";
    for( ; lo != up; ++ lo) {
        os << *lo << ' ';
    }
    os << "}";
    // count
    os << " C:" << data.count(what) // count hits
        << "\n";
}
int main() {
    set data{ 10, 20, 30, 40, 50, 60 };
    search(data, 20, cout); // 20? in:yes. found:20 lo:20 up:30 range:{20 } C:1
    search(data, 25, cout); // 25? in:no. lo:30 up:30 range:{} C:0
    search(data, 10, cout); // 10? in:yes. found:10 lo:10 up:20 range:{10 } C:1
    search(data, 60, cout); // 60? in:yes. found:60 lo:60 up:- range:{60 } C:1
    search(data, 5, cout); // 5? in: no. lo:10 up:10 Range:{} C:0
    search(data, 99, cout); // 99? in: no. lo:- up:- Range:{} C:0
}
```

Listing 24.56 These are the “set” search functions.

If the combination of return values seems strange to you, you might remember them by their typical use cases:

- `lower_bound` finds the search element or—if not present—its best insertion position.
- `equal_range` returns the range of all found elements, which is particularly useful for the multi... variants of the containers. If no element fits, the range is empty.
- `upper_bound` returns the first element that no longer fits.

It helps me to imagine that the found boundaries lie *between* the elements, not on the elements themselves. In [Figure 24.11](#), I have drawn the returned iterators at the *beginning* of the element. If you imagine it that way, the terms `upper_bound` and `lower_bound` suddenly fit perfectly.

I mostly use `find` or `lower_bound` myself, and sometimes `equal_range` with `multiset`. `upper_bound` is used less frequently, but you can find an example in [Listing 24.58](#).

The search functions are also available in the `<algorithms>` header for all compatible containers, especially for sequence containers. However, they must be sorted beforehand. You can find more about this in [Chapter 25](#).

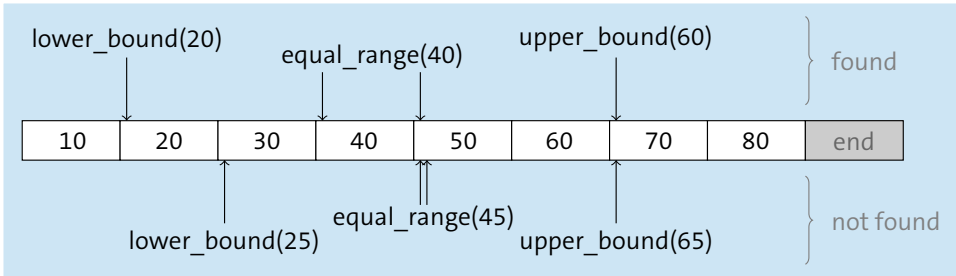


Figure 24.11 The search functions define ranges.

Searching without a Specific Key

The parameters of `find` and `find`-related functions naturally require a key as a parameter for the search. But what if generating a key is very costly or even impossible? Or the search keys are slightly different in form from the stored keys? In that case, you can use a different type as a parameter for the search key. It is important that the keys are equivalent (with respect to the search).

```
// https://godbolt.org/z/zEsxd8e7G
#include <string>
#include <set>
#include <iostream>
#include <tuple> // tuple, tie
using std::string; using std::set; using std::cout; using std::tie;
struct Hobbit {
    string firstName;
    string lastName;
    Hobbit(const string v, const string n) : firstName{v}, lastName{n} {}
};
struct CompLastName {
    bool operator()(const Hobbit& x, const Hobbit& y) const { // normal <
        return tie(x.lastName, x.firstName) < tie(y.lastName, y.firstName);
    }
    using is_transparent = std::true_type; // allowed for find
    bool operator()(const Hobbit& x, const string& y) const { // for find etc.
        return x.lastName < y;
    }
    bool operator()(const string& x, const Hobbit& y) const { // for find etc.
        return x < y.lastName;
    }
};
int main() {
    using namespace std::literals; // allow "...s
    set<Hobbit, CompLastName> hobbits;
```

```
hobbits.emplace( "Frodo", "Baggins" );
hobbits.emplace( "Sam", "Gamgee" );
auto f1 = hobbits.find( Hobbit{"Frodo", "Baggins"} ); // whole key
if(f1 != hobbits.end()) {
    cout << "found: " << f1->firstName << '\n';      // Frodo
}
auto f2 = hobbits.find( "Gamgee"s );                // equivalent key
if(f2 != hobbits.end()) {
    cout << "found: " << f2->firstName << '\n';      // Sam
}
}
```

Listing 24.57 You can search with nonidentical keys if they are equivalent.

In [Listing 24.57](#), to search for "Gamgee"s, you do not need to create a whole search element of type `Hobbit`. For this, the configured comparison functor `CompLastName` must have certain properties. For the normal comparisons that `set` uses internally, and the search for `find(Hobbit{"Frodo", "Baggins"})`, the first operator() is defined, which takes two `Hobbit` arguments. For the search for "Gamgee"s, the other two overloads are there, which take a `Hobbit` and a string. However, with the line

```
using is_transparent = std::true_type;;
```

it should also be indicated that these overloads can be used for search functions. Thus a string can be used directly for the search, and no entire `Hobbit` is created.

Note that in this way you can only search if the alternative comparison functions do not imply a different order than the main comparison operator. You cannot expect the set to be kept sorted in one order, but then search with a comparison operator that would sort things differently. This is what is meant by an *equivalent* comparison function.

Deleting

For deletion, you have `erase` and `clear` available. With `extract`, you remove an element, which you usually transfer to another set with `insert`:

■ Deleting all elements

You delete all elements with the `clear()` method.

■ Deleting a single element

If you pass an iterator to the `erase` method, the element at that position will be deleted. Alternatively, you can also specify an element as a parameter, which will be deleted if it exists. Both take $\log n$ time. The return value is the number of deleted elements.

■ Transferring an element

With `extract`, you remove a single element from the set. The return value is a “handle” for the removed node, which you can use for other purposes, such as transferring it to another similar set using `insert`.

■ Deleting multiple of elements

With a pair of iterators as arguments, `erase` can delete the elements between them.

All iterators of nondeleted elements retain their validity.

```
// https://godbolt.org/z/dh8Yqx6c4
set data{ 10, 20, 30, 40, 50, 60, 70 };
auto lo = data.lower_bound(35);
auto up = data.upper_bound(55);
data.erase(lo, up);      // deletes all numbers between 35 and 55
cout <<= data;           // Output: 10 20 30 60 70
lo = data.lower_bound(20);
up = data.upper_bound(60);
data.erase(lo, up);      // deletes including 60, because up points to 70
cout <<= data;           // Output: 10 70
auto n = data.erase(69); // deletes nothing
cout << "Number of removed elements: "<< n << '\n'; // Output: Number ... 0
n = data.erase(70);      // deletes one element
cout << "Number of removed elements: "<< n << '\n'; // Output: Number ... 1
cout <<= data;           // Output: 10
```

Listing 24.58 “`erase`” deletes one or more elements.

24.7.4 “map”

A map is used to quickly find values using keys. Each key (Key) in a map is associated with exactly one target (Target). The map is related to the set in that the rules for the elements of set apply to the keys of map:

- Keys in a map are always sorted.
- No key appears more than once in a map.
- Searching for a key is guaranteed to take at most $\log n$ steps.

Much of what applies to set also applies to map, so [Section 24.7.3](#) can provide additional information.

However, there is a big difference from set: the individual elements in the map are a pair, specifically a `pair<const Key, Target>`. When I refer to this pair, I use the term *Element* or *elem* in the context of map.

Value Is Ambiguous in “map”

Sometimes the key and target pair is also called the *value*. Elsewhere, *value* refers to what I named target. To avoid this misunderstanding, I strive to avoid the term *value* in the context of `map`.

If you come from another language like Python or Perl, I want to immediately draw your attention to a pitfall you might stumble over, because the C++ `map` behaves differently than the Python dictionary or the Perl hash. If you access the `map` with `operator[]` and the key element you are accessing does not yet exist, then you *create an entry* in the `map`. This also applies if you use `[]` on the right side of an assignment and think you are only reading.

```
// https://godbolt.org/z/zWa58dYYq
#include <map>
#include <iostream>
using std::map; using std::cout;
int main() {
    map<int, char> alpha;
    cout << alpha.size() << '\n';           // 0 naturally
    if( alpha[5] == '3' ) { /* ... */ }
    cout << alpha.size() << '\n';           // now 1
    char x = alpha[99];                      // works
    cout << alpha.size() << '\n';           // and now 2
}
```

Listing 24.59 Using `[]` may create an entry as a side effect.

This is a miraculous increase in value! But if you are familiar with it, you can handle it: if a key that you pass to `operator[]` does not yet exist in the `map`, then a new entry is inserted, where a new target is created with its default constructor—in the preceding case, `char '\0'`.

This is useful when you want to use `[]` in longer expressions as it never fails. If this behavior bothers you, use `at` instead. If the key does not exist, an exception is thrown. With `find` or `contains`, you can determine without an exception whether a key is present in the `map`.

If you do not like this implicit behavior of `operator[]`, use `insert_or_assign` instead for inserting into a `map` or `unordered_map`.

The examples in this section for `map` should be considered embedded in the code of [Listing 24.60](#). Note my comment on `<=>` in the box that followed [Listing 24.12](#).

```
// https://godbolt.org/z/K834js588
#include <map>           // the main thing
#include <iostream>       // for output
#include <string>         // often used for key or target
using std::map; using std::cout; using std::string;
template<typename Key, typename Trg, typename Comp>
std::ostream& operator<<=(std::ostream&os, const map<Key,Trg,Comp>&data) {
    for(auto &e : data) os << e.first << ":" << e.second << ' ';
    return os << '\n'; // '<=<' instead of '<<' for line break
}
int main() {
    // Example code here
}
```

Listing 24.60 This is the template for the example listings in this section on “map”.

The Elements and Their Properties

The elements of `map` are `pair<const Key,Target>`—that is, a pair consisting of a key and target. For the key, the rules of the elements as in `set` apply, as well as the associated rules for comparison and sorting. By default, the compiler uses a less-than `<` to internally arrange the keys in a tree and keep them constantly sorted.

If you don't like `<`, you can specify your own comparison function as an additional template parameter.

This also works for custom data types, but you can also simply overload `operator<=>` or `operator<`.

```
// https://godbolt.org/z/W6Mfnxj1W
#include <cstdio> // toupper, tolower
// ...
auto comp = [](char a, char b) { return toupper(a) < toupper(b); };
map<char,int,decltype(comp)> lets(comp); // as template parameter and argument
lets['a'] = 1;
lets['B'] = 2;
lets['c'] = 3;
lets['A'] = 4; // overwrites position 'a'
cout <<= lets; // Output: a:4 B:2 c:3
struct Comp { // Functor
    bool operator()(char a, char b) const { return toupper(a) < toupper(b); }
};
map<char,int,Comp> lets2; // here the template parameter is sufficient
```

Listing 24.61 You can also provide a custom comparison function for a “map”.

When you define a functor—that is, a class with an `operator()`—it is sufficient to specify this class as a template parameter. When creating the `map`, an instance of the functor is then created using its default constructor. With a lambda, you already have an instance that is callable.

Initialize

As a template parameter for `map`, you specify the types for `Key` and `Target`—for example:

```
map<int, string>           // int as Key, string as Target
map<Person, size_t>       // Person to their age?
map<unsigned, unsigned>    // Key and target types can also be the same
map<int, shared_ptr<Image>> // a pointer as the target type is quite common
```

Optionally, you can specify the types of the *comparator* for `Key` (see previous section) as well as an *allocator*. If you specify one, you can optionally also provide the usual constructor arguments.

The following variants are commonly used as constructor arguments for initialization:

- Default constructor creates an empty container
- Iterator pair copies a range from another container or stream
- `map` copies an entire, other `map`
- Initializer list initializes with explicit values
- Constructor with the *from_range* tag and a range from C++23

For filling via an initializer list, you must consider that each element consists of a pair. So you must also specify pair elements here. Fortunately, pairs can also be created by a two-element initializer list, so the construct still looks compact, as shown in [Listing 24.62](#).

```
// https://godbolt.org/z/vW4xsMPq7
using std::pair; using std::make_pair;
namespace literal_p { // better to put user-defined literals in a namespace
constexpr pair<char, char> operator "" _p(const char* s, size_t len) {
    return len>=2 ? make_pair(s[0], s[1]) : make_pair( '-', '-' );
} }
struct Q {
    char a_; int n_;
    Q(char a, int n) : a_{a}, n_{n} {}
    operator pair<const char, int>() { return make_pair(a_, n_); }
};
// ...
// explicit pairs:
map<int, int> nums { pair<int, int>(3,4), make_pair(7,8), make_pair(11,23) };
map num2 { pair<int, int>(6,1), make_pair(5,2) };
```

```
// implicit pairs from initializer lists:
map<int, char> numch{{1, 'a'}, {2, 'b'}, {3, 'c'}};
map<int, int> nums3 { {7, 2}, {9, 4} };
using namespace literal_p;
map<char, char> pmap { "ab"-p, "cd"-p, "ef"-p }; // detour via custom literal
cout <<= pmap; // Output: a:b c:d e:f
map<char, int> qmap{Q('a', 1), Q('b', 2), Q('c', 3)}; // implicit conversions
cout <<= qmap; // Output: a:1 b:2 c:3
```

Listing 24.62 The initializer list must contain “pair” elements.

As you can see in the case of `qmap`, the elements of the initializer list can indeed be implicitly converted to a `pair<const Key, Target>`. This is quite useful for compact notation.

Assignment

Similar to `set`, you can reinitialize a `map` with `operator=`. There are overloads for copy and move.

An `assign` or `assign_range` method is also not available in `map`, but like in all other containers, there is `swap`.

Insert

Just like with a `set`, you add new elements with `insert`, `emplace`, `emplace_hint`, and from C++23 with `insert_range`. There are also `operator[]` and `insert_or_assign`, which may also insert new elements under certain circumstances.

No matter how you insert, iterators pointing to the `map` retain their validity.

Inserting a Single Element with Automatic Positioning

In `map`, a single element consists of a pair of key and target. You then specify this as a parameter. As with `set`, `insert` and `emplace` do not overwrite in this case.

```
// https://godbolt.org/z/Y1ebrxb8T
map<int, string> zip2plc;
zip2plc.insert(std::make_pair(94103, "San Francisco"));
zip2plc.emplace(10001, "New York");
cout <<= zip2plc; // Output: 10001:New York 94103:San Francisco
map<string, int> plc2zip;
plc2zip.emplace("New York", 10001);
plc2zip.emplace("New York", 10002); // does not overwrite
cout <<= plc2zip; // Output: New York:10001
```

Listing 24.63 You specify a single new element as a pair.

Automatic Insertion and Overwriting

With `operator[]`, you can search for a key. The return value is a *reference* to the target. If there is no entry for the key yet, it will be newly created. For this, a new target is then created with its default constructor `Target{}`. If `[]` appears on the left side of an assignment, you can immediately overwrite the newly created target with a new target.

```
// https://godbolt.org/z/7h3vsWvYG
map<string,int> dwarves;
dwarves.emplace("Fili", 2859);
cout << dwarves["Fili"] << '\n'; // Output: 2859
cout << dwarves["Dori"] << '\n'; // newly created. Output: 0
dwarves["Kili"] = 2846;           // newly created and immediately overwritten
cout << dwarves["Kili"] << '\n'; // Output: 2846
cout <<= dwarves;                // Output: Dori:0 Fili:2859 Kili:2846
```

Listing 24.64 Automatically create and immediately overwrite with “`operator[]`”.

Sometimes it is more efficient to use `insert_or_assign(key, value)`. If key already exists in the map, the same thing happens as with `operator[]`. But if the entry is new, an empty element is not first created at the position of key; instead, value is inserted directly. In addition, the return value is a pair with information about whether and where the insertion took place.

Inserting a Single Element with a Position Hint

`insert` optionally takes an iterator as the first parameter, which represents the probable insertion position of the new element. If this is correct, the insertion happens on average in constant time. This also applies to `emplace_hint`, where the position hint is an additional parameter.

Inserting Multiple Elements

Using a pair of iterators, you can insert multiple elements as with `set`, and from C++23 onward, you can also use `insert_range`.

Accessing

Because the elements of a map consist of `pair<const Key,Target>`, all iterators from functions and methods refer to such a pair. This applies to iterators as parameters and in return values.

You can access the pair of an iterator `it` with `*it` or dereference directly with `it->`. Thus, you get a constant reference to the key with `it->first`, and a mutable reference to the target with `it->second`.

The fact that the second element of the pair is not `const` means that you can assign a new value to the target. The key then refers to the new target.

```
// https://godbolt.org/z/8948besY8
map<string,string> data { {"John","Wayne"}, {"Cary","Grant" }, };
cout <<= data; // Cary:Grant John:Wayne
data.rbegin()->second = "Travolta"; // Overwrite target
cout <<= data; // Cary:Grant John:Travolta
```

Listing 24.65 You can change the value of a target.

If you use the range-based `for`, you get the pair as `e` directly and can access the key and target with `e.first` and `e.second`.

Access via Iterators

As with all ordered associative containers, you get bidirectional iterators back with `[c][r]begin()` and `[c][r]end()`.

```
// https://godbolt.org/z/16T5dob75
map<char,int> data { { 'a',1}, { 'b',2}, { 'c',3} };
for(auto it=data.rbegin(); it!=data.rend(); ++it) { // backwards
    cout << it->first << ':' << it->second << ' '; // dereference with ->
}
cout << '\n'; // Output: c:3 b:2 a:1
for(auto &e : data) { // forwards, uses begin() and end()
    cout << e.first << ':' << e.second << ' '; // pair, element access with .
}
cout << '\n'; // Output: a:1 b:2 c:3
```

Listing 24.66 Iterators of “map” are of type “pair”.

Access via Search

You can search for keys in the map using `find`, `upper_bound`, `lower_bound`, and `equal_range`. You get an iterator to the found element or `end()` back, as described with `set`. With `count`, you can find out if a key occurs zero times or one time; with `contains`, you get that information as a `bool` since C++20.

Index-Like Access

As a special feature among associative containers, ordered or unordered, `map` has the `operator[]` and `at` methods, which you know from `vector` and `array`. This allows you to use a map as if it had an index. Unlike its counterparts, the index is not of type `size_t`, but of type `Key`. Therefore, the argument for `operator[]` and `at` is a value of this type. As shown in [Listing 24.59](#) and [Listing 24.64](#), `operator[]` automatically inserts missing elements if the key is not found. `at`, on the other hand, throws an `out_of_range` exception in such cases. If you have a `const` map, you cannot use `operator[]`, because it is not marked as `const` due to the automatic insertion. This is demonstrated in [Listing 24.67](#).

```
// https://godbolt.org/z/Eqsjex4Ta
string search7(const map<int,string> &data) {
    return data[7];           // ⚡ non-const method on const parameter
}
string search5(const map<int,string> &data) {
    auto it = data.find(5);    // not automatically inserting
    return it==data.end() ? string{} : it->second;
}
// ...
map<int,string> dwarfs{ {1,"one"}, {3,"three"}, {5,"five"}, {7,"seven"} };
cout << search7(dwarfs) << '\n';
cout << search5(dwarfs) << '\n'; // Output: five
```

Listing 24.67 You cannot use “operator[]” on a “const map”.

Deleting

Deleting in map is hardly different from deleting in set:

- **Deleting all elements**
clear() clears the map completely.
- **Deleting a single element**
erase deletes a single element by iterator or with a key.
- **Transferring a single element**
extract removes an element that you want to put elsewhere with insert.
- **Deleting multiple of elements**
Using a pair of iterators for erase, you delete multiple elements.

After deletion, iterators in the container are still valid if their elements were not affected by the deletion.

Specialty: “operator[]”

Among the associative containers, only map offers the index operator operator[]. This greatly simplifies the use of map, and source code with maps looks compact and readable.

However, it has a peculiarity that you really need to remember. It automatically inserts a new element into the container if there is no entry for the key yet. I have already demonstrated this with several examples, such as in [Listing 24.59](#), [Listing 24.64](#), and [Listing 24.67](#), but I cannot emphasize it enough.

24.7.5 “multiset”

A multiset, like a set ([Section 24.7.3](#)), contains elements that are also keys, which you can quickly search for. The difference is that elements can also occur multiple times. So,

for example, if you insert the value 7 three times into a `multiset<int>`, you will get three sevens when iterating through it. `insert` and `emplace` will always add one or more elements. The behavior of the various search functions adapts accordingly: The returns of `lower_bound`, `upper_bound`, and `equal_range` now point to the edges of the range with matching elements. `find` returns any matching element. `erase`, when called with a (key) element, deletes all matching elements, not just one.

`merge` removes all elements from the source, instead of leaving duplicates behind.

If you want to use `multiset`, you need `#include <set>`.

The examples for `multiset` should be considered embedded in [Listing 24.68](#).

```
// https://godbolt.org/z/aMj87neG6
#include <set>      // multiset
#include <iostream>
using std::multiset; using std::cout;
template<typename Elem, typename Comp>
std::ostream& operator<<=(std::ostream&os, const multiset<Elem,Comp>&data) {
    for(auto &e : data) {
        os << e << ' ';
    }
    return os << '\n'; // '<=<' instead of '<<' for line break
}
int main() {
    // Example code here
}
```

Listing 24.68 This is the template for the example listings in this section on “multiset”.

Note my comment on `<=<` in the box that followed [Listing 24.12](#).

The Elements and Their Properties

As with `set`, the elements in the `multiset` are the keys by which the container is kept sorted. Here too, you can overload either `operator<=<` or `operator<` for your element type, or provide your own comparator as an optional template argument.

To make it clear, so that there are no misunderstandings, especially not in comparison to the `unordered_multiset`: all elements in the `multiset` are stored in a sorted tree, including duplicates. You can see this exemplified in [Figure 24.12](#). It is *not* the case that duplicates are stored in a linked list or something similar. This means that even with many or only duplicate entries, the performance of a `multimap` does not degrade. The `multimap` is just as fast with *n identical* elements as it is with *n different* ones. The key property of `multiset` is that its efficiency is guaranteed and not easily degraded—unlike the `unordered_...` variants if they have poor hash functions. In [Listing 24.75](#), you can see

how `set`, `multiset`, and `unordered_multiset` perform in different scenarios. One conclusion is that many identical elements do not degrade the performance of an `unordered_multiset`, but many different ones do with a poor hash function.

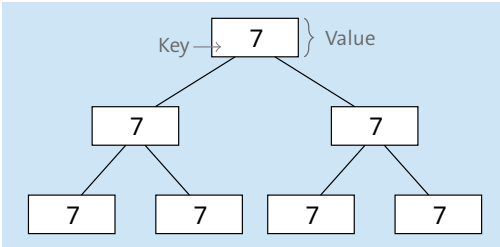


Figure 24.12 Even duplicate elements are stored in a “multiset” in a sorting tree.

An alternative is the `flat_multiset` adapter from C++23, which stores the elements sorted in a vector. Check this especially with many containers or very small elements.

Initialize

A `multiset` has exactly the same construction options as a `set`:

- Without arguments via default constructor
- A range from another container using a pair of iterators
- Another `multiset` for copying
- An initializer list
- From C++23, a constructor with the `from_range` tag

In many cases, the compiler can deduce the template parameters from the constructor arguments if you are using C++17.

If you choose an initializer list or a range from another container, you get a container that contains the elements sorted and retains the duplicate entries.

```
// https://godbolt.org/z/aqGTKdbMP
#include <vector>
// ...
multiset msinit{1,2,2,3,1};           // sorted at initialization
cout <<= msinit;                     // Output: 11223
std::vector in{ 7,7,7,7,7,7,7 };
std::set srange( in.begin(), in.end() ); // set removes duplicates
cout << srange.size() << ": " << *srange.begin() << '\n'; // Output: 1: 7
multiset msrange( in.begin(), in.end() ); // multiset retains duplicates
cout <<= msrange;                     // Output: 7777777
```

Listing 24.69 Entries are sorted, and duplicates are retained.

Assignment

With `operator=`, you can reassign the contents of `multiset` just like a `set`, either by copy or by move. The `swap` method is also used here to implement common idioms.

Insert

You insert into a `multiset` using the same overloads as in a `set`, with one exception: where `set` returns a `pair<iterator, bool>` to indicate via `second` whether an insertion actually took place, `multiset` only returns the iterator. Finally, `multiset` always inserts, regardless of whether the element already exists or not:

- **Inserting a single element with automatic positioning**

Use `insert` to insert a single value by copy, or `emplace` to create the element directly in place. The return is a single iterator pointing to the inserted element—unlike in `set`, where you get a pair with the second element being a `bool`. `multiset` inserts in $O(\log n)$ time, regardless of whether the element is already present or not.

- **Inserting a single element with a position hint**

If you provide an iterator along with the value to be inserted, `multiset` also tries to interpret this iterator as a hint for insertion. If you do not want to pass a value but want to create it in place, use `emplace_hint`. You also know both from `set`, and here too the amortized time complexity is $O(1)$ if the hint is correct, and $O(\log n)$ if not.

- **Inserting multiple elements**

Like `set`, `multiset` also understands inserting multiple elements at once if you provide an initializer list or multiple via an iterator pair. From C++23 onward, there is also `insert_range`. As expected with `multiset`, all elements are transferred, including duplicates.

As with `set`, all iterators of the container remain valid after an insertion in `multiset`.

Accessing

The interface of `multiset` for accessing is identical to that of `set`. Only the duplicate elements in the container need to be accounted for in the returns. Either they return any matching element or encompass the entire range of all matching elements:

- **Access via iterators**

You use the variants of `begin()` and `end()` that you know from `set` to obtain bidirectional iterators.

- **Access via search**

If `find` finds the search element in the container, then it is *any* of the matching elements. This is different with `lower_bound`, `upper_bound`, and `equal_range`, because with the specification already introduced in `set`, you get the area boundaries of all matching elements of the container. Here, `count` is interesting because the number of found elements has more informational value in this case.

■ Searching without a specific key

You can save the creation of a concrete search key during the search if you equip a suitable comparison functor with `is_transparent`, as shown for `set` in [Listing 24.70](#).

The following listing offers some examples of accessing a `multiset`.

```
// https://godbolt.org/z/rWGrhbYa7
#include <string>
#include <iterator> // distance
#include <ranges>    // subrange
struct Person {
    std::string name;
    friend bool operator<(const Person &a, const Person &b) {
        // only first letter
        return a.name.size()==0 ? true
            : (b.name.size()==0 ? false : a.name[0] < b.name[0]);
    }
};
// ...
multiset data{ 1, 4,4, 2,2,2, 7, 9 };
auto [from1, to1] = data.equal_range(2);
cout << "Number of 2s: "
    << std::distance(from1, to1) << '\n';    // Output: Number of 2s: 3
auto [from2, to2] = data.equal_range(5);
cout << "Number of 5s: "
    << std::distance(from2, to2) << '\n';    // Output: Number of 5s: 0
multiset<Person> room{
    {"Karl"}, {"Kurt"}, {"Peter"}, {"Karl"}, {"Ken"}};
auto [p, q] = room.equal_range(Person{"K"});
for(auto& p : std::ranges::subrange(p,q)) { // C++20 range or simple loop
    cout << p.name << ' ';
}
cout << '\n'; // Output: Karl Kurt Karl Ken
```

Listing 24.70 The “multiset” search functions find the range of matching elements.

In the example `room`, with an appropriate `operator<`, you can indeed consider actually different elements in a `multiset` as equal. Here, for example, only the first letter of the name is compared. When I then search with `raum.equal_range(Person{"K"})`, all names with `K` fall within the returned range. Note that the defined `operator<` is also used for internal sorting, so only the first letters of the names are used for sorting. This is why the unsorted “K...” names appear in the search results.

The return value of `equal_range` is a pair, and since C++17, it can be assigned to two new variables using a *structured binding*. Before C++17, you would use `auto x` and then `x.first` or `x.second`.

Deleting

As with `set`, you delete with `erase` or `clear`:

- **Deleting all elements**

With `clear()`, you remove everything.

- **Deleting or transferring a single element**

`erase` deletes either a single element if you provide an iterator, or all elements that match the given search key.

`extract` does that too if you want to insert it elsewhere afterward.

- **Deleting multiple of elements**

With two iterators as parameters, you delete multiple elements as with `set`.

All iterators for nondeleted elements retain their validity.

24.7.6 “multimap”

The `multimap` has two relatives: The commonality with `map` is that each entry consists of a key and a target. From `multiset` comes the property that keys can occur multiple times. A `multimap<int, string>` could therefore store both the target “sieben” and the target “seven” under the key 7, as shown in [Figure 24.13](#).

I will therefore be brief and refer you to [Section 24.7.4](#) and [Section 24.7.5](#) for more information.

A crucial difference in the interface of `multimap` compared to `map` is that `multimap` offers neither `operator[]` nor `at`.

The examples in this section for `multimap` should be considered embedded in [Listing 24.71](#).

```
// https://godbolt.org/z/bxW63jPor
#include <map>           // the main thing
#include <iostream>      // for output
#include <string>        // often used for key or target
using std::multimap; using std::cout; using std::string;
template<typename Key, typename Trg, typename Cmp>
std::ostream& operator<<=(std::ostream&os, const multimap<Key,Trg,Cmp>&data){
    for(auto &e : data) {
        os << e.first << ":" << e.second << ' ';
    }
    return os << '\n'; // '<<=' instead of '<<' for line break
}
int main() {
    // Example code here
}
```

Listing 24.71 This is the template for the listings in this section on “multimap”.

Note my comment on `<<=` in the box that followed [Listing 24.12](#).

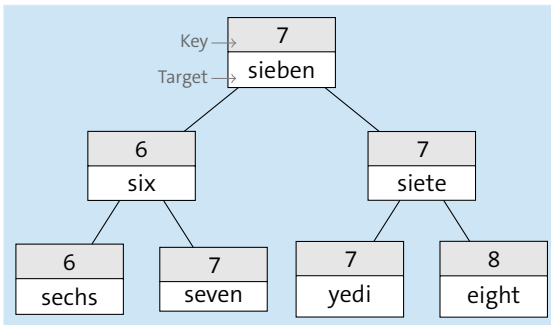


Figure 24.13 There can be duplicate keys.

The Elements and Their Properties

Just like with `map`, the elements stored in a `multimap<Key,Target>` are of type `pair<const Key,Target>` for key and target.

The keys are again kept sorted using `operator<`. However, you can also provide your own comparator.

Initialize

You initialize a `multimap` just like a `map`, except that duplicate keys are accepted and not ignored.

You have a choice among the following:

- The default constructor without arguments
- A pair of iterators for copying another range
- An initializer list, which in this case consists of pairs
- Another `multimap` for copying
- From C++23, the constructor with the `from_range` tag

The following listing shows an example of initializing a `multimap`.

```
// https://godbolt.org/z/6no9dW64d
multimap<int,int> std::make_pair(3,4) }; // multimap<int,int>
using namespace std::literals; // for ""s
multimap<int,string> numlang{
    {7,"seven"s}, {6,"six"s},
    {7,"siete"s}, {6,"sechs"s},
    {7,"seven"s}, {7,"yedi"s},
    {8,"eight"s} };
cout <<= numlang; // Output: 6:six 6:sechs 7:seven 7:siete 7:seven 7:yedi 8:eight
```

Listing 24.72 All entries end up in the “multimap”.

As you can see, duplicate keys are preserved. Duplicate targets are naturally possible with `map` anyway. The keys are kept sorted at all times, as you can see from the output.

Assignment

The `multimap`, like `map`, has an `operator=` for copying or moving from another `multimap` afterwards. With `swap`, you can quickly exchange the contents of two `multimaps`.

`merge` transfers all elements between two `multimaps` and does not leave duplicates in the source as with `map`.

Insert

You can insert new elements into the `multimap` with `insert`, `emplace`, `emplace_hint`, and from C++23 also `insert_range`. Unlike `map`, elements are *always* inserted, even if the new key already exists. What does not exist compared to `map` is `operator[]` for automatic insertion:

- **Inserting a single element with automatic positioning**

As with `map`, you can use `insert` to add a new `pair<Key,Target>` to the `multimap`. With the `emplace` method, you provide `Key` and `Target` directly as parameters. Unlike with `map`, these two do not return a `bool` indicating the success of the insertion, but only an iterator with the position of the inserted element. This is demonstrated in [Listing 24.73](#).

- **Inserting a single element with a position hint**

As with `map`, you can provide `insert` with an additional iterator as a position hint or use `emplace_hint` in the same way. If the hint is correct, then the insertion happens on average in $O(1)$.

- **Inserting multiple elements**

The `insert` of the `multimap` can also receive a pair of iterators, just like with `map`. Starting from C++23, you can also use `insert_range`.

The following listing shows how to insert elements into a `multimap`.

```
// https://godbolt.org/z/dj68ej97r
using namespace std::literals; // for ""s

multimap<int,string> numlang{};
numlang.insert( std::make_pair(7, "seven"s) );
numlang.insert( std::pair<int,string>(7, "sieben"s) );
numlang.emplace( 7, "yedi"s );
cout <<= numlang; // Output: 7:seven 7:sieben 7:yedi
```

Listing 24.73 “insert” and “emplace” in the “multimap”.

As you can see, identical keys are always inserted last among identical ones.

As usual with ordered associative containers, iterators of a `multimap` are not affected by insert operations and remain valid.

Accessing

As with `map`, the iterators of the `multimap` reference a `pair<const Key, Target>`. You can overwrite the `Target`—unless, of course, you have obtained a `const_iterator`.

Unlike with `map`, there is no index-like access with `operator[]` or `at`:

- **Access via iterators**

As with `map`, you get bidirectional iterators with `multimap` using the `begin()` and `end()` methods and their `const` and reverse counterparts.

- **Access via search**

You search in a `multimap` just like in a `map` using `find`, `upper_bound`, `lower_bound`, and `equal_range` to look for keys. The interface is identical, except that `find` finds any matching key, and the rest can actually return an entire range of multiple identical keys. `count` gives you the number of keys found, `contains` from C++20, whether it is contained at all.

Deleting

There is hardly any difference when deleting compared to `map`, except for `erase` by key search:

- **Deleting all elements**

With `clear()`, you delete all elements of a `multimap`.

- **Deleting or transferring a single element**

`erase` deletes a single element by iterator, as usual with `map`.

With `extract`, you delete to insert elsewhere with `insert`:

- **Deleting a single key**

If you specify a key, then all entries of that key are removed in the `multimap`. This is nominally the same with `map`, as there is only a maximum of one key, but I particularly emphasize it for the `multimap`.

- **Deleting multiple elements**

`erase` with a pair of iterators refers to multiple elements to be deleted.

The following listing demonstrates deletion—with iterators of nondeleted elements of the container retaining their validity.

```
// https://godbolt.org/z/TbvKTn4Px
multimap<char,int> vals{ {'c',1}, {'c',8}, {'g',1},
    {'c',1}, {'a',7}, {'a',1}, {'c',2}, };
cout << vals;           // Output: a:7 a:1 c:1 c:8 c:1 c:2 g:1
```

```
vals.erase( 'c' );           // deletes all 'c's
cout <<= vals;               // Output: a:7 a:1 g:1
vals.erase(vals.begin());    // deletes only one of the 'a's
cout <<= vals;               // Output: a:1 g:1
```

Listing 24.74 “erase” with a key can delete multiple elements.

24.8 Only Associative and Not Guaranteed

In this section, Cont or C stands for one of the container types and cont for an instance of it. As *key type* I use Key or K, as *target type* Target or T, and the *value type* pair<K,T> is Value or V. *Iterator types* are abbreviated as It.

Container	Description
unordered_set	Contains a unique set of elements
unordered_map	Unique keys refer to a target value
unordered_multiset	Is a set of elements that can occur multiple times
unordered_multimap	Keys can occur multiple times

Table 24.8 Profile: The associative unordered containers.

These containers do not store their keys sorted, but *hash* them. This has two effects: first, the elements are not in a specific order when you iterate; and second, hashing is often faster than sorting.

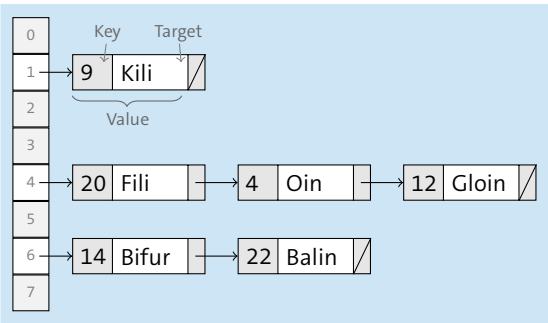


Figure 24.14 An “unordered_map” internally groups the keys by hash code. In the “unordered_set”, the values themselves are the keys.

24.8.1 Hash Tables

While map, set, and their multi... variants store their elements in a permanently sorted tree, the unordered variants use lookup tables called *hash tables*.

By using the tree, `map` and `set` guarantee an upper bound on the time consumed during searches. Hashing is usually faster, but it cannot provide this guarantee. It is important to know that in extreme cases it can be much slower. Whether hashing works well depends on many factors; for example, you might have an inefficient hash function (without realizing it) or receive unfriendly data (which is beyond your control).

Hashing, Tables, Buckets, and Collisions

When an element is to be inserted or searched, the container first calculates a *hash value* for the element. This is a numerical value from a very large range—for example, between zero and `UINT_MAX`. This hash value determines the position of the element in the *hash table*. However, because this table cannot have `UINT_MAX` entries, it is much smaller; let's say that it has 1,024 entries. The position in the hash table is determined by the hash value modulo 1024, which is the remainder of the division. Elements whose hash values are, for example, 7, 1031, or 2055 are stored under the same table entry. Typically, these elements are stored as a list.

The table entries are also called *buckets*. When checking if a given element is in the container, the hash value quickly leads to its bucket. The elements in this bucket all have the same hash value (modulo table size), but they may differ in terms of equality `==`, which is called *collision*. Therefore, further searching within the bucket is necessary, and this can take a long time if there are many elements in the bucket. That is why the container makes a great effort to keep the number of collisions low. For example, the table size is adjusted repeatedly as the number of elements in the container grows. If everything runs normally, most buckets should contain only one element. Another important factor in keeping the number of collisions low is the quality of the hash function.

Use the unordered variants only if you first need that extra bit of speed and second have control over the keys of the container—ensuring that they do not become “unfriendly” in terms of frequency and hash. The ordered associative containers are more robust in this regard.

You must exercise great care when selecting the hash function. Although unfavorable input data can also be disastrous, a poorly chosen hash function leads to performance catastrophes more quickly. In [Listing 24.75](#), you can see several different containers with different data and, in the case of the `unordered_multiset`, with different hash functions in use.

Providing Your Own Hash Function

Whenever the standard library needs a hash function for the `T` data type, it uses the `std::hash<T>` functor. This is predefined for all built-in types. For custom data types, you must provide a specialization. This is discussed further ahead in the context of [Listing 24.78](#).

Unordered associative containers also use these functors. Alternatively, you can also provide them with your own functor as a constructor argument. This is significantly easier than injecting a functor into the `std` namespace.

```
// https://godbolt.org/z/1MYhxb7rq
#include <set> // set, multiset
#include <unordered_set> // unordered_set, unordered_multiset
#include <iostream>
#include <string>
#include <chrono> // Time measurement
using std::cout;
using namespace std::chrono;

long long millisSince(steady_clock::time_point start) {
    return duration_cast<milliseconds>(steady_clock::now()-start).count();
}

constexpr size_t ITERATIONS = 100'000;
template<typename Cont, typename Gen>
    requires std::invocable<Gen, size_t> && // C++20 concept
    requires(Gen gen, size_t n) {{gen(n)} -> std::same_as<int>;} &&
    std::same_as<typename Cont::value_type, int>
void timeStuff(std::string name, Cont data, Gen genNum) {
    cout << name << "...";
    auto start = steady_clock::now();
    for(size_t idx=0; idx<ITERATIONS; ++idx) {
        data.insert( genNum(idx) );
    }
    cout << " " << millisSince(start) << " ms" << std::endl;
}

int same(size_t) { return 7; } // always generates the same number
int scatter(size_t n) { return int(n); } // generates different numbers
struct BadHash { // the worst possible hash function as a functor
    size_t operator()(int) const { return 1uz; }
};

int main() {
    std::multiset<int> m{};
    timeStuff("multiset", m, &same);
    timeStuff("multiset", m, &scatter);
    std::set<int> s{};
    timeStuff("set", s, &same);
    timeStuff("set", s, &scatter);
    std::unordered_multiset<int> um{};
```

```
timeStuff("unordered_multiset same", um, &same);
timeStuff("unordered_multiset scatter", um, &scatter);
std::unordered_multiset<int, BadHash> umb{};
timeStuff("unordered_multiset same badHash", umb, &same);
timeStuff("unordered_multiset scatter badHash", umb, &scatter);
}
```

Listing 24.75 Never operate unordered associative containers with a bad hash function.

The output from the preceding listing on my computer is, for example, as follows:²

```
multiset      same      ... 72 ms
multiset      scatter   ... 66 ms
set           same      ... 8 ms
set           scatter   ... 74 ms
unordered_multiset same   ... 27 ms
unordered_multiset scatter ... 29 ms
unordered_multiset same   badHash... 24 ms
unordered_multiset scatter badHash... 255419 ms
```

I have `timeStuff` exemplarily provided with C++20 concepts. For `Gen` it is required that `gen` must be a functor that can be called with a `size_t` and whose return must be an `int`. For `Cont`, the `value_type` must be an `int`.

You can make the following observations:

- The `multiset` almost doesn't care whether the elements are `same` or `scatter`.
- `set` is naturally ultrafast with `same` because it ignores duplicates and thus constantly holds only one element.
- With a reasonable hash function, it doesn't matter to `unordered_multiset` whether the values are the same or scattered.
- Only with a poor hash function like `badHash` does the performance catastrophically degrade when different elements are sorted into the same bucket.

Performance also heavily depends on the implementation of the compiler and the associated standard library. Here is the same program on the same machine, but compiled with a different compiler (Clang++ 9.0):³

```
multiset      same      ... 57 ms
multiset      scatter   ... 49 ms
set           same      ... 6 ms
set           scatter   ... 71 ms
unordered_multiset same   ... 112802 ms
```

² Intel i7-3520M CPU, 2.90GHz, gnu c++ 6.0, std=gnu++1y.

³ Repeated in 2023 with Clang++ 16. It produced equally good results as g++.

```
unordered_multiset scatter      ... 28 ms
unordered_multiset same    badHash... 119559 ms
unordered_multiset scatter badHash... 117607 ms
```

Note that the performance of an unordered associative container can also degrade if you have a not-so-bad hash function but unfavorable data. Only the ordered variants provide guarantees, even though the unordered ones are faster in the optimal case.

You can see in the modified example ahead how little fun a bad hash function can be. Here I vary the number of inserted elements. This gives you a sense of how quickly the runtime can increase depending on the number of elements.

```
// https://godbolt.org/z/cc3KohPrY
#include <unordered_set>    // unordered_set, unordered_multiset
#include <iostream>
#include <string>
#include <chrono>           // Time measurement
using std::cout;
using namespace std::chrono;
long long millisSince(steady_clock::time_point start) {
    return duration_cast<milliseconds>(steady_clock::now()-start)
        .count();
}
struct BadHash { // the worst possible hash function as a functor
    size_t operator()(int) const { return 1uz; }
};
void timeStuff(size_t iters) {
    std::unordered_multiset<int, BadHash> data{};
    cout << iters << "...";
    auto start = steady_clock::now();
    for(size_t idx=0; idx<iters; ++idx) {
        data.insert( (int)idx );
    }
    cout << " " << millisSince(start) << " ms" << std::endl;
}
constexpr size_t LIMIT = 20'000;
int main() {
    size_t iters = 100;
    while(iters < LIMIT) {
        timeStuff(iters);
        iters *= 2; //double
    }
}
```

Listing 24.76 Twice as many elements with a poor hash function means four times the runtime.

The output speaks volumes. If the amount of input doubles, then the runtime quadruples:

```
100... 0 ms
200... 1 ms
400... 5 ms
800... 17 ms
1600... 69 ms
3200... 274 ms
6400... 1104 ms
12800... 4260 ms
```

This is due to the fact that all elements, even if they are not the same, end up in a single bucket. And that becomes a long list that must be traversed entirely to determine where the element belongs.⁴

24.8.2 Commonalities and Differences

In [Table 24.9](#), you can see the main characteristics that distinguish the unordered associative containers from each other. The following notes apply to the * entries:

- **Key type**

The ordered associative containers can use almost any type as a key, as long as its values can be compared with `equal ==` and equipped with a hash function.⁵ To do this, carefully overload `hash(Key)`.

- **Memory layout**

Internally, a table of the hash values of the keys is stored. In this process, multiple different keys can be thrown together into a bucket. To search for a specific key, the correct bucket is determined, which then may need to be examined for all its elements (see the “Hashing, Tables, Buckets, and Collisions” box at the beginning of this section).

- **Searching**

Normally, elements can be found in constant time $O(1)$. However, if the hash values of the keys clump (and in the worst case are all the same), the performance degrades catastrophically to $O(n)$ per search.

Spaceship `<=>` and Equality `==`

The rules for when `operator<=>` is sufficient and when you need to implement `operator==` are somewhat complicated. As a rule of thumb, if you can implement `operator<=>` with `= default`, you should do so and not define any additional operators, especially not `==`.

⁴ Most implementations use a list, but it could also be something else.

⁵ You must be able to create, remove, and copy them.

However, if you define an operator<=> yourself (i.e., do not write = default), then you also need a self-defined operator==. Why? Because it is quite possible that for your class, *equal* is much faster to compute than *less*. And if you implement <=> yourself, you are putting in the greater effort for *less*, which would be wasted if it were only for checking *equal*.

For example, if you compare the strings "abc"s and "abcd"s, you can quickly determine that the strings are not == based on their length, but for a complete <=> you need to look at all the characters.

So, if the compiler looks for operator== for unordered containers, it only considers operator<=> if it is = default. If it is user-defined, you need a separate operator==.

If you only want to make your data type suitable for unordered containers, you can limit yourself to operator==.

Property	unordered_set	unordered_map	unordered_multiset	unordered_multimap
Unique keys	Yes	—	yes	—
Keys to target values	—	Yes	—	Yes
Key type*	hash, ==	hash, ==	hash, ==	hash, ==
Dynamic size	Yes	Yes	Yes	Yes
Overhead per element	Yes	Yes	Yes	Yes
Memory layout*	Hash table	Hash table	Hash table	Hash table
Insert, remove	Usually very fast			
Search*	Usually very fast			
Sorted	—	—	—	—
Iterators/ranges	Forward	Forward	Forward	Forward
Algorithms*	Unsorted			

Table 24.9 Properties of unordered associative containers.

For all unordered_... containers, you must choose and implement the hash function for the key type very carefully; otherwise, you will get poor performance. Unfortunately, it is anything but trivial to prove or automatically verify the behavior of a hash function. Especially with the unpredictability of incoming data, a good hash function is tricky.

When implementing a hash function for your data, try to use other hash functions from the standard library as much as possible.

Even if you have a perfect hash function, you also need to ensure that you do not receive “degenerated” data at runtime. Unfortunately, you rarely have full control over the exact data.

For these two reasons, I can only recommend that you always choose the ordered associative containers with their guaranteed runtime behavior as your first choice of data structure. Only if you need the last bit of performance and can react to unforeseen data should you choose an unordered variant.

Fill Level and Reordering

With these containers, it can happen that the number of buckets needs to be adjusted when inserting. This is called the *load factor*, and it uses the `size()/bucket_count()` ratio. You can query the load factor with `load_factor()`. When it reaches the `max_load_factor()`, an automatic increase occurs. If an automatic increase occurs, a complete reordering of the elements takes place—a rehash. All iterators into the container lose their validity. The order of the elements relative to each other is almost certainly different after such a step than before.

Similar to `vector`, you can never be sure if iterators are still valid after an insertion. In addition, you can (also similar to `vector`) use a call to `reserve` to ensure that this does not happen up to a given limit. And you can pass a number to the constructor of these containers when creating them.

You could also change the `max_load_factor`, but you normally shouldn't. Initially, this is set to 1.0. Do you remember that you can pass the initial bucket count to the constructor? A `max_load_factor` of 1.0 means that a rehash occurs when the number of elements exceeds this value. With a `max_load_factor` of 0.5, this would happen at half that number.

24.8.3 Methods of Unordered Associative Containers

I am not always very precise or detailed with the function signatures here to save space and maintain clarity. If I don't include the parentheses with `method`, there are usually multiple overloads:

- **Constructors**—`Cont()`, `Cont(Compare)`, `Cont{...}`, `Cont(It, It)`, `Cont(from_range, ...)`

The default constructor `Cont()` creates an empty container. With a pair of iterators from another compatible container, you can copy its contents. You can specify an initializer list that your container should contain at the beginning. For an `unordered_map`, this means you need to specify nested `{...}`, as each element must initialize a pair with two parameters—for example, `unordered_map<int, string>{{2, "two"s}, {3, "three"s}}`. Starting from C++23, there are also `from_range` constructors.

- **Copy and move**—`C(const C&)`, `C(C&&)`, `C& op=(const C&)`, `C& op=(C&&)`, `swap`
You can pass another `Cont` as an argument to the copy and move constructors and operators. All associative containers implement efficient `swap(Cont& other)`.
- **Destructor**—`~Cont()`
All containers have destructors that remove the contained elements. Note that this does not apply if you pack raw pointers into containers.
- **Iterate forward**—`begin()`, `end()`; and **backward**—`rbegin()`, `rend()`
You can iterate here just like with sequential containers. However, the order in which you obtain the elements is random (or, more precisely, *arbitrary*) and does not necessarily correspond to the order of insertion or any sorting. The iterators of the `unordered_map` variants reference a `pair<K, T>`.
- **Element access**—`T& operator[] (Key)`, `T& at(Key)`
Only `unordered_map` has these methods and behaves like `map`.
- **Size**—`size()`, `empty()`, `max_size()`, `reserve`
With `size()`, you get the current number of elements in the container; `empty()` is true if this number is zero. With `max_size()`, you get the library-specific maximum size of the container. Because containers of this family grow in stages similar to a vector, they also have a `reserve` method to control the timing.
- **Modification**—`clear()`, `erase`, `insert`, `insert_or_assign`, `insert_range`, `emplace`, and `merge`
These methods work like those in sequence containers. In maps, an element consists of a `pair<K, T>`, which you can insert using `insert` and `emplace`. `merge` does not produce a sorted result (of course), unlike in lists and ordered containers. `insert_range` is available from C++23. Only `unordered_map` has `insert_or_assign`.
- **Search**—`count`, `find`, `contains`, `equal_range`
These methods correspond to those of the sorted containers. `count` counts how often a key occurs, `find` returns an iterator to a found key. `contains` checks for existence since C++20. `equal_range` returns a pair of iterators between which the found keys are located.

24.8.4 “unordered_set”

An `unordered_set` has an interface similar to that of a `set`; see [Section 24.7.3](#).

When you iterate, you only have forward iterators available. The *forward* here does not refer to an order in which you receive the elements, but rather to the fact that you can only apply `++` and not `--` to the iterators.

You should imagine the examples in this section for `unordered_set` embedded in the code of [Listing 24.77](#).

The order of outputs may vary for unordered containers depending on the implementation of the standard library.

Note my comment on <<= in the box that followed [Listing 24.12](#).

```
// https://godbolt.org/z/38TbjGhn6
#include <unordered_set>
#include <iostream>
using std::unordered_set; using std::cout; using std::ostream;
template<typename Elem, typename Cmp>
ostream& operator<<=(ostream&os, const unordered_set<Elem,Cmp>&data){
    for(auto &e : data) {
        os << e << ' ';
    }
    return os << '\n'; // '<=' instead of '<<' for line break
}
int main() {
    // Example code here
}
```

Listing 24.77 This is the template for the example listings on “unordered_set”.

Element Properties

Unlike set, the elements of unordered_set must support the operator==. Things that you consider “equal” for the purpose of the container must return true for operator== (const Elem &a, const Elem &b). Alternatively, you can specify a KeyEqual functor as a template parameter during the definition, which will then be used for key comparison instead.

In addition, you now need a functor that calculates a hash value from Key. To do this, write a specialized class template std::hash<Key> with a size_t operator()(const Key &k) const method. Make sure that you utilize the range of size_t as much as possible for all conceivable k. Only then will the container have good performance. If you cannot or do not want to overload the function, specify a template parameter for Hash when defining the container.

```
// https://godbolt.org/z/TY5x54efE
#include <unordered_set>
#include <iostream>
#include <vector>
#include <string>
using std::string; using std::unordered_set; using std::cout;
struct Word {
    string word_;
    size_t row_;
```

```

    Word(const string &word, size_t row)
        : word_{word}, row_{row} {}
    friend bool operator==(const Word& a, const Word &b)
        { return a.word_ == b.word_; } // ignores row
};
namespace std {
template<> struct hash<Word> { // ignores row
    std::hash<string> stringHash;
    std::size_t operator()(const Word &w) const {
        return stringHash(w.word_);
    }
}; }
struct ExactWordHash { // includes row
    std::hash<string> sHash;
    std::hash<size_t> iHash;
    bool operator()(const Word& a) const {
        return sHash(a.word_) ^ iHash(a.row_);
    }
};
struct ExactWordEqual { // includes row
    bool operator()(const Word& a, const Word &b) const {
        return std::tie(a.word_, a.row_) == std::tie(b.word_, b.row_);
    }
};
int main() {
    std::vector input {
        Word{"a",0}, Word{"rose",0},
        Word{"is",1}, Word{"a",1}, Word{"rose",1},
        Word{"is",2}, Word{"a",2}, Word{"rose",2}, };
    // Use overloads
    unordered_set<Word> words( input.begin(), input.end() );
    cout << words.size() << '\n'; // Output: 3
    // Use custom functors
    unordered_set<Word, ExactWordHash, ExactWordEqual> poem(
        input.begin(), input.end() );
    cout << poem.size() << '\n'; // Output: 8
    // Hash as Lambda
    auto h = [](const auto &a) { return std::hash<string>{}(a.word_); };
    unordered_set<Word, decltype(h)> rose(input.begin(), input.end(), 10, h);
    cout << rose.size() << '\n'; // Output: 3
}

```

Listing 24.78 An “unordered_set” with custom comparison and hash function.

For the first example with `words`, I use the approach of defining the operations where the normal search expects them. The default for the template arguments are `std::hash<Key>` and `std::equal<Key>`. The latter uses the `operator==(Key, Key)` free function in its implementation. Both are defined here: `operator==` is visible as a free function via the `friend` method, and `hash<Word>` is a template specialization of `template<typename Key> struct hash` in the `std` namespace. It should be noted that you should not arbitrarily define things in the `std` namespace: it is acceptable to make specializations like these in `std`, but you must not define new things in `std` arbitrarily; you would end up with nonportable code.

The second `poem` example is somewhat more direct, as it does not use the poorly visible built-in mechanisms. Here I have specified the functors to be used as template arguments. This makes it somewhat clearer, because especially for the hash function, you cannot simply define a `friend` function within your own data type.

As usual, you do not necessarily have to use a functor; a callable object is sufficient. Finally, you see with `rose` that I passed the lambda `h` as the hash function to the constructor. However, I also have to pass its type as a template argument using `decltype(h)`. In addition, the order of the constructor arguments requires that I also pass the number of *buckets* of the container before the hash function—hence the `10` here.

Initialize

As always, when defining and initializing, you need to define two things: the template parameters and the constructor arguments. Since C++17, the compiler can often deduce the template parameters from the constructor arguments, allowing you to omit the part with the angle brackets, `<Key>`.

As template arguments for the `unordered_set`, at least the key—that is, element type `Key`—is required—for example, `unordered_set<int>`:

- `Key`, the key type for searching
- Optional `Hash`, the hash functor for `Key`; default is `std::hash<Key>`
- Optional `KeyEqual`, comparison operator for `Key`; default is `std::equal_to<Key>` and thus `operator==`
- Optional `Allocator`, memory allocator for the elements

Optionally, specify a hash functor, a comparison functor for keys, and an allocator. As shown in [Listing 24.78](#), the standard library uses the `operator==` free function if you do not provide a comparison functor. That means, for custom data types, that a `friend bool operator==` or an `operator<=>` with `= default` is sufficient. Thus, in most cases, specifying a comparison functor is not necessary. In practice, the hash functor remains, which you occasionally need to define. You will also see examples of this in [Listing 24.78](#). The easiest way is with a lambda and `decltype`:

```
auto h = [](const auto &a) { return ... ; };
unordered_set<Key,decltype(h)> data( ... h );
```

You can provide constructor arguments similar to set and the other containers, as follows:

- No arguments
- A range from another container using a pair of iterators
- An initializer list
- The tag *from_range* and a range, if you use C++23
- Another `unordered_set` to copy from

Except for the last two variants, all optionally take a `size_t`, with which you can set the initial size of the hash table.

This looks, in a brief example, as shown in [Listing 24.79](#).

```
// https://godbolt.org/z/1j6PMojWr
#include <set>
template<typename Key>
std::set<Key> sorted(const unordered_set<Key> &data)
{ return std::set<Key>(data.begin(), data.end()); }
// ...
// without arguments
unordered_set<int> empty{};
cout <<= empty;    // Output:
// initializer list
unordered_set data{1,1,2,2,3,3,4,4,5,5}; // duplicates are not included
cout <<= data;      // Output approximately: 5 4 3 2 1
// copy
unordered_set copy(data);
cout <<= copy;      // Output approximately: 5 4 3 2 1
// Range
auto so1 = sorted(data);
unordered_set range(std::next(so1.begin()), std::prev(so1.end()));
cout <<= range;     // Output approximately: 2 3 4
```

Listing 24.79 These are the ways to initialize an “`unordered_set`”.

Note that the order of the elements may be different in your case. To make sure that I skip 1 and 5 when copying the last example, I copied the `unordered_set` data into the sorted container `so1` using `sorted()`. For this, I defined that helper function.

Assignment

For assigning new content, overloads of `operator=` are available, allowing copying and moving. With `swap`, some programming idioms can be implemented efficiently. Both are already known from `set`.

Insert

As with `set`, you insert with `insert`, `insert_range`, `emplace`, or `merge`:

■ Inserting a single element with automatic positioning

Inserting a single element with `insert` or `emplace` takes $O(1)$ time in the best case and $O(n)$ time in the worst case. While `insert` copies the new element, with `emplace` you provide the constructor arguments for creating it in place. Note that, as with `set`, an element is not replaced if it is already contained in the `unordered_set`. Whether something was actually inserted can be determined from the `bool` in the returned pair. [Listing 24.80](#) shows an example.

■ Inserting a single element with a position hint

As with `set`, you can speed up an insertion a bit if you already know where the element belongs. To do this, use `insert` with a position iterator or `emplace_hint`. However, the savings are of a completely different nature than usual. With `unordered_set`, the standard library can save the calculation of the hash as well as the cumbersome traversal through the bucket list. A single key comparison is enough to determine whether the element needs to be inserted. [Listing 24.80](#) contains an example of this at the end. What seems somewhat unnecessary with `unordered_set` makes more sense with `unordered_multiset`. If the position is not correct, the additional time required is minimal.

■ Inserting multiple elements

Like all containers, `unordered_set` can also insert an entire area using a pair of iterators or an entire initializer list. Starting with C++23, there is also `insert_range`. With `merge`, you can retrieve multiple elements from another `unordered_set`, but only the elements that are not already present in the current one are transferred.

The next listing shows basic insertion into an `unordered_set`.

```
// https://godbolt.org/z/js3MeoccM
unordered_set<int> data;
auto res1 = data.insert( 5 );           // Insertion by copy
if(res1.second) cout << "yes, 5 now inside\n"; // that works
auto res2 = data.emplace( 5 );         // In-place insertion
if(res2.second) cout << "second 5 now inside\n"; // that doesn't work
auto res3 = data.insert(res1.first, 6 ); // with position hint
// res3 is just an iterator without bool
cout << *res3 << '\n';                 // definitely a 6
```

Listing 24.80 Insertion into an “`unordered_set`”

All insertion operations potentially invalidate iterators in the container. However, if you are sure that no reordering of elements has taken place, then iterators remain valid—for example, because you previously called the `reserve` method appropriately.

Accessing

Unlike `set`, the *unordered associative containers* only provide *forward iterators*, as I have already shown in [Table 24.2](#). So they are not suitable for sorting at all:

■ Access via iterators

You get an iterator to the beginning with `begin()` and `cbegin()`, and an iterator past the end of the container with `end()` and `cend()`. When you then iterate over the elements with `++it`, `next(it)`, or `advance`, their order is arbitrary but stable across multiple passes, provided you have not modified the container in the meantime.

■ Access via search

For unordered associative containers, certain search functions, like those available for ordered ones, do not make sense; two searches for different keys do not really enclose a range. So you only have `find`, `count`, `contains`, and `equal_range`. In `unordered_set`, `count` returns zero or one. `equal_range` either encompasses a range with exactly one element or returns `end()` twice.

A search without a concrete key is not possible with unordered associative containers because the search requires a hash value, and a hash value requires a specific key.

Deleting

The methods for deletion are the same as with `set`, though the interface is slightly restricted:

■ Deleting all elements

You clear `unordered_set` with `clear()`.

■ Deleting or transferring a single element

You can use an iterator or a key to search for `erase` as a parameter. The return value is an iterator to the element after the deleted one. With `extract`, you remove an element that you can insert elsewhere with `insert`.

■ Deleting multiple elements

You can specify multiple elements with two iterators for `erase`, but the unspecified order within `unordered_set` makes this only partially meaningful unless you use `begin()` and `end()` as parameters, which you can better achieve with `clear()`.

Iterators to nondeleted elements retain their validity.

Furthermore, deleting elements does not affect the order of the remaining elements among themselves. This means you can delete multiple elements within a loop without worrying about whether a delete action has changed the order of the remaining elements so that you might not see some elements anymore.

```
// https://godbolt.org/z/hee331WGe
unordered_set nums{ 1,2,3,4,5,6,7,8,9,10 };
cout <<= nums;           // Output similar to: 9 1 2 3 4 5 6 7 8 10
auto it = nums.begin();
while(it != nums.end()) {
    if(*it % 2 == 0) {      // even number?
        it = nums.erase(it); // Remaining elements do not change order
    } else {
        ++it;
    }
}
cout <<= nums;           // Output similar to: 9 1 3 5 7
```

Listing 24.81 Deleting preserves the order of the remaining elements.

Even if the order of the output may look different on your computer, you will get the same order in the second output as in the first, just without even numbers.

The call to `erase` returns an iterator to the next element. If `erase` is not to be called, then `++it` ensures the iterator moves forward.

Specialty: By the Bucketful

You can take a close look at the contents of the individual buckets:

- With `bucket_count()`, you get the current number of buckets in the container. The number cannot exceed `max_bucket_count()`.
- `begin(int)` and `end(int)` provide you with iterators to the elements of a single bucket.
- Beforehand, you can check the number of elements in a single bucket with `bucket_size(int)`.
- With `int bucket(Key)`, you can find out which bucket an element would end up in.

I find it difficult to imagine a meaningful application for this deep interface. However, I can imagine that such insights can be useful when you have to deal with the disadvantage of unordered containers, which do not offer any time guarantees. I can only provide you with an example that lists the contents of the individual buckets of an `unordered_set` with curious eyes.

```
// https://godbolt.org/z/39v4aqqd1
// Fill with 100 values
unordered_set<int> d{};
d.rehash(10);           // try to have 10 buckets
d.max_load_factor(100.0); // 100 elements per bucket are okay
cout << "Bucket count: " << d.bucket_count() << '\n';
```

```

for(int x : std::ranges::iota_view{0,100}) { // C++20 iota(): 0,1,2,...,99
    d.insert(x);
}
// output
for(int b = d.bucket_count()-1; b>=0; --b) {
    cout << "Bucket "<<b<<": ";
    for(auto it=d.begin(b); it!=d.end(b); ++it)
        cout << *it << ' ';
    cout << '\n';
}

```

Listing 24.82 You can access “unordered_set” by bucket.

For me, the attempt with `rehash(10)` resulted in getting 11 buckets. Often, prime numbers are a good idea. Otherwise, the output on my computer looked as follows:

```

Bucket 10:98 87 76 65 54 43 32 21 10
Bucket 9:97 86 75 64 53 42 31 20 9
Bucket 8:96 85 74 63 52 41 30 19 8
Bucket 7:95 84 73 62 51 40 29 18 7
Bucket 6:94 83 72 61 50 39 28 17 6
Bucket 5:93 82 71 60 49 38 27 16 5
Bucket 4:92 81 70 59 48 37 26 15 4
Bucket 3:91 80 69 58 47 36 25 14 3
Bucket 2:90 79 68 57 46 35 24 13 2
Bucket 1:89 78 67 56 45 34 23 12 1
Bucket 0:99 88 77 66 55 44 33 22 11 0

```

You can see here for each individual bucket which elements it contains.

Specialty: Reserve Space

Each insertion can potentially cause all elements of the `unordered_set` to be rearranged. Then iterators lose their validity. Similar to `vector`, you can influence when this happens and thus control the timing. The exact mechanism is explained in [Table 24.9](#). Here are the involved functions:

- `float load_factor()` is the average number of elements per bucket.
- When `load_factor` exceeds `float max_load_factor()`, a reorganization of the elements occurs. You can change the value yourself with `max_load_factor(float)`. In most implementations, this value is initially 1.0.
- `rehash(size_t b)` sets the number of buckets to at least `b`, but never less than `size()/max_load_factor()`.
- `reserve(size_t c)` is the best way to ensure that `c` elements are well accommodated in the container with respect to the current `load_factor`.

In [Listing 24.82](#), I demonstrated `rehash` and `reserve`. `reserve(size_t c)` takes into account a previously set `max_load_factor(float)`. It corresponds to the call of `rehash(ceil(c/max_load_factor()))`.

24.8.5 “unordered_map”

The `unordered_map` shares large parts of its interface with `map` ([Section 24.7.4](#)) and often exhibits similar behavior to `unordered_set` ([Section 24.8.4](#)).

This implies that it is possible to link target values to keys and retrieve them rapidly. Of particular note is that this is usually achieved in $O(1)$ time, which is optimal. Regrettably, this may not always be guaranteed, and in extreme cases the process could take $O(n)$ time.

As with `map`, you have `operator[]`, `at`, and `insert_or_assign` available. Otherwise, you search with forward iterators.

The examples in this section for `unordered_map` should be considered embedded in the code of [Listing 24.83](#). However, note my comment on `<<=` in the box that followed [Listing 24.12](#).

```
// https://godbolt.org/z/T5KKWx4z4
#include <unordered_map>
#include <iostream>
using std::unordered_map; using std::cout;
template<typename K, typename T>
std::ostream& operator<<=(std::ostream&os, const unordered_map<K,T>&data) {
    for(auto &e : data) {
        os << e.first << ":" << e.second << ' ';
    }
    return os << '\n'; // with operator<<= followed by a newline
}
int main() {
    // Example code here
}
```

Listing 24.83 This is the template for the example listings on “unordered_map”.

The Elements and Their Properties

The elements of an `unordered_map` are `pair<const Key,Target>`. For Key, you need two things:

- You need a hash functor, which is either `std::hash<Key>` or specified by you as a template argument.
- Key must be comparable with `==`, meaning it must be defined, or you must provide a functor for this.

For custom data types, the easiest way is to define `operator==` as friend or `operator<=>` with `=` default and also provide a custom hash functor. A short example is shown in [Listing 24.84](#).

```
// https://godbolt.org/z/cdeGW6a3P
#include <unordered_map>
#include <iostream>
#include <string>
using std::string; using std::unordered_map; using std::cout;
struct City {
    string name_;
    explicit City(const string &name) : name_{name} {}
    auto operator<=>(const City &b) const = default;
};
struct CityHash {
    std::hash<string> sHash;
    size_t operator()(const City& a) const {
        return sHash(a.name_);
    }
};
int main() {
    unordered_map<City, string, CityHash> cty{
        {City{"San Francisco"}, "CA"},
        {City{"Austin"}, "TX"},
        {City{"Miami"}, "FL"},
    };
    cout << cty[City{"San Francisco"}] << '\n'; //Output: CA
}
```

Listing 24.84 A custom data type as a key in an “unordered_map”.

The spaceship operator `<=>` with `=` default ensures that the compiler can use it for `==` and thus it can be used by the `unordered_map`.

The `CityHash` functor requires a `size_t operator()(const City&) const` method. In it, I use the hash functor provided by the standard library for `string`.

Initialize

When defining an `unordered_map`, you need at least the template arguments for `Key` and `Target`.

You can only omit them if you use `pair` for initialization; unfortunately, the compiler cannot help with nested initialization lists.

The rest is optional:

- **Key**—the key type for searching
- **Target**—the type to which the keys should refer
- **Optional Hash**—the hash functor for Key, default is `std::hash<Key>`
- **Optional KeyEqual**—the comparison operator for Key; by default `std::equal_to<Key>` and thus `operator==`
- **Optional Allocator**—memory allocator for the elements

Besides the types, you can use various constructor overloads. Most of the following options also have the possibility to provide a hash function, a comparison operator, or an allocator:

- The default constructor without arguments to create an empty container
- An iterator pair for copying from another container or stream
- An initializer list with initial values
- From C++23, *from_range* with a range
- Another `unordered_map` for a complete copy or move

A peculiarity of all unordered associative containers is that in the first three variants, an optional `size_t` count can still be passed, which sets the minimum size of the initial hash table.

Assignment

Like all unordered containers, you can use `operator=` for both copying and moving to reinitialize the container. With `swap`, you have a tool for very quickly exchanging all data with another `unordered_map`.

Insert

When inserting, there is no difference in the interface compared to `map`. Only the run-time guarantees are different as they are consistently between constant and linear per inserted element:

- **Inserting a single element with automatic positioning**
Use `insert(pair<Key,Target>)` or `emplace(Key,Target)` to insert a new element. Be careful here as well: the container will not overwrite if the key already exists.
- **Automatic insertion and overwriting**
Alternatively, you can also use `operator[]` for insertion, but this will first create an empty Target. If you use this on the left side of an assignment, you will get a reference to this Target back, which you can then reassign. `insert_or_assign` saves the creation of an empty value.

- **Inserting a single element with a position hint**

There are also `insert` and `emplace_hint` with an additional iterator parameter in `unordered_map`.

- **Inserting multiple elements**

You can copy multiple elements with an iterator pair or an initializer list using `insert`, or use `insert_range` starting from C++23.

With `merge`, you can extract elements from another `unordered_map` that do not yet exist here.

Iterators in the `unordered_map` retain their validity like in the `unordered_set` as long as the number of buckets does not need to be adjusted. If you want to have control over this, use `reserve()` beforehand, for example.

Accessing

When accessing elements with the `it` forward iterators of the `unordered_map`, they refer to `pair<const Key, Target>`. This means you cannot change the `it->first` key, but you can change the target value, `it->second`.

Otherwise, the interface is roughly equivalent to that of a normal `map`. Only with the search functions do you need to limit yourself to what all unordered associative containers offer:

- **Access via iterators**

With `begin()` and `end()` as well as `cbegin()` and `cend()`, you get forward iterators that you can use, for example, in the range-based `for` loop.

- **Access via search**

You have `find`, `count`, `contains`, and `equal_range` available for searching. Because an `unordered_map` only knows unique keys, `find()` probably makes the most sense here. As with `unordered_set`, you unfortunately cannot search without creating a complete key as you would with `map`.

- **Index-like access**

The special features of `unordered_map` are, as with `map`, the `operator[]` and the `at` method. With both, you get a reference to the target value of an existing entry back. The difference between the two is that `operator[]` inserts a new empty element if none exists, while `at` throws an exception of type `std::out_of_range` if the key does not exist.

Deleting

- **Deleting all elements**

Clear the `unordered_map` with `clear()`.

- **Deleting or transferring a single element**

As with `map`, `erase` from `unordered_map` deletes either by key or by iterator. With `erase`, you delete an element to immediately reinsert it elsewhere with `insert`.

- **Deleting multiple elements**

With a pair of iterators, `erase` deletes multiple elements.

Iterators to nondeleted elements remain valid through deletion operations. Moreover, the relative order of the remaining elements in the container does not change, allowing you to delete elements within a loop without disrupting the loop.

Specialty: “operator[]”

Like `map`, the `unordered_map` offers the `operator[]` for easy reading and writing. No other `unordered` associative container has this access capability.

Specialty: By the Bucketful

The `unordered_map` also has an interface that allows you to gain precise insights into its internal workings. You can use `bucket_count()` to get the number of buckets and `[c]begin/end(int)` iterators to access the contents of individual buckets. In addition, `bucket_size(int)` and `int bucket(Key)` are available to you.

Specialty: Reserve Space

To maintain control over when a reordering of elements occurs during insertion, you can influence this mechanism with several methods, similar to `unordered_set`. `load_factor` and `max_load_factor` are the interfaces for load factor, and with `rehash` and `reserve` you can directly influence the size of the hash table.

24.8.6 “unordered_multiset”

The `unordered_multiset` inherits its properties from `unordered_set` ([Section 24.7.3](#)) and from `multiset` ([Section 24.7.5](#)).

It usually finds its keys optimally fast in $O(1)$, but unfortunately cannot guarantee this and then needs $O(n)$ time in the worst case, which is why the `ordered_multiset` might be the better choice if you want to avoid nasty surprises.

As with `multiset`, each key can be stored multiple times. One of the typical use cases is that only a part of the stored objects actually serves as keys. There is an example in [Listing 24.85](#).

Element Properties

The elements of an `unordered_multiset` are simultaneously its keys, as with `set`. And as with `unordered_set`, you need `equality ==` for the keys and a hash function. For custom

data types, the best option is to define operator<=> with = default or, if that is not possible, operator==.

```
// https://godbolt.org/z/Ed88znPn9
#include <unordered_set> // unordered_multiset
#include <iostream>
#include <string>
using std::string; using std::unordered_multiset; using std::cout;
struct City {
    string name_;
    explicit City(const string &name) : name_{name} {}
    auto operator<=>(const City &b) const = default;
};
struct Entry { string city_; int zip_; };
struct EqEntry {
    bool operator()(const Entry&a, const Entry&b) const {
        return a.city_==b.city_;
    }
};
struct HashEntry {
    std::hash<string> sHash;
    size_t operator()(const Entry& a) const {
        return sHash(a.city_);
    }
};
int main() {
    unordered_multiset<Entry, HashEntry, EqEntry> directory{
        {Entry{"New York", 10001}},
        {Entry{"New York", 10002}},
        {Entry{"New York", 10003}},
        {Entry{"Chicago", 60601}},
        {Entry{"Chicago", 60602}},
    };
    const Entry search{"New York", 0}; // ZIP code does not matter in search
    cout << "New York has " << directory.count(search) << " ZIP codes.\n";
    cout << "The ZIP codes of New York are:\n";
    auto [where, until] = directory.equal_range(search);
    while (where != until) {
        cout << " " << where->zip_ << '\n';
        ++where;
    }
}
```

Listing 24.85 You can also use only part of your object as a key.

The output, of course, is as follows:

New York has 3 ZIP codes.

The ZIP codes of New York are:

10001

10002

10003

The `EqEntry` and `HashEntry` functors, which are specified as template arguments, ensure that all entries with the same `city_` are considered equal. It would not be enough to define only the hash functor or only the equality; both operations must match as otherwise the entries will get mixed up and not end up in the correct buckets. Therefore, you must carefully select and program the two functors.

For example, if you include more elements in equality than in the hash function, you will not find an entry with `count(search)`, but only with `count(Entry{"New York", 10001})`. It may be that this is what you wanted, but then your hash function is not optimal, and you should have included `zip_`:

```
struct EqEntry {
    bool operator()(const Entry&a, const Entry&b) const {
        return tie(a.city_, a.zip_) == tie(b.city_, b.zip_); // ⚡ Really?
    }
};

struct HashEntry {
    std::hash<string> sHash;
    size_t operator()(const Entry& a) const {
        return sHash(a.city_);
    }
};
```

It would be worse the other way around, if equality covered fewer elements than the hash functor:

```
// https://godbolt.org/z/cveT8Ej7q
struct EqEntry {
    bool operator()(const Entry&a, const Entry&b) const {
        return a.city_==b.city_;
    }
};

struct HashEntry {
    std::hash<string> sHash;
    std::hash<int> iHash;
```

```

size_t operator()(const Entry& a) const {
    return sHash(a.city_) ^ iHash(a.zip_); // ⚡ too many elements
}
};

```

Now you probably won't find a single entry anymore. Different New Yorks, which are actually equal by `==`, have landed in different hash buckets and are thus distributed throughout the entire container. You still have the chance to find individual entries like `Entry{"New York", 10001}`. However, because two New York entries can also randomly end up in the same bucket, even that is not guaranteed. The result is data garbage.

Initialize

Regarding the template parameters, you declare an `unordered_multiset` just like an `unordered_set`, meaning you need to specify the entry or key type and can then also specify the hash functor, key comparison functor, and allocator.

However, starting from C++17, the compiler can help you by often deducing the template parameters from the constructor arguments.

The possibilities for initialization via constructor are similar to `unordered_set`, except that duplicates are preserved. Here too, except for the copy constructor, all options can accept an additional parameter to specify the initial size of the hash table:

- The default constructor without arguments initializes the container as empty.
- A pair of iterators from another container or stream copies the range defined by them.
- In an initializer list, you can explicitly specify the initial elements.
- You can completely copy another `unordered_multiset`.
- Starting from C++23, there is also a constructor with the *from_range* tag.

Note in the following example that duplicates are preserved.

```

// https://godbolt.org/z/bvehE8hse
#include <unordered_set> // unordered_multiset
#include <vector>
#include <iostream>
using std::unordered_multiset; using std::cout; using std::ostream;
template<typename Elem>
ostream& operator<<=(ostream&os, const unordered_multiset<Elem>&data) {
    for(auto &e : data) { os << e << ' '; } return os << '\n'; }
int main() {
    // without arguments
    unordered_multiset<int> empty(1000); // initial size of the hash table
    cout <<= empty; // Output:

```

```
// Initialization list; duplicates are included:
unordered_multiset data{ 1,1,2,2,3,3,4,4,5,5 };
cout <<= data;      // Output approximately: 5 5 4 4 3 3 2 2 1 1
// Copy
unordered_multiset copi(data);
cout <<= copi;      // Output approximately: 5 5 4 4 3 3 2 2 1 1
// Range
std::vector in{1,2,3,10,20,30,10,20,30,1,2,3};
unordered_multiset rang(in.begin()+3, in.end()-3);
cout <<= rang;      // Output approximately: 30 30 20 20 10 10
}
```

Listing 24.86 These are the ways to initialize an “unordered_multiset”.

Of course, the order of elements in the output can vary from system to system.

Assignment

As with all other containers, you can use `operator=` to copy or move data from another `unordered_multiset`. Also, `swap` is defined for quick data exchange.

Insert

Insertion happens in `unordered_multiset` just like in `multiset`. Unlike an `unordered_set`, every insertion is successful because duplicates are also stored:

- **Inserting a single element with automatic positioning**

With `insert`, you copy a new element into the container, while with `emplace`, you create it in place.

- **Inserting a single element with a position hint**

If you provide an iterator with `insert`, it may save the computation of the hash value and the traversal of elements within a bucket. The position hint should then point to an element that is equal to the new element. If you do not provide a correct element, the advantage is lost, but the element still ends up in the right place. The variant for creating in place is `emplace_hint`.

- **Inserting multiple elements**

`insert` also accepts a pair of iterators or an initializer list for inserting multiple elements. From C++23, there is also `insert_range`. With `merge`, you retrieve all elements from another container, including duplicates.

If you have not made provisions with `reserve`, you must assume that all iterators to the container have become invalid after an insertion operation.

Accessing

You have forward iterators available, as you can see in [Table 24.2](#):

- **Access via iterators**

With `begin()` and `end()` as well as `cbegin()` and `cend()`, you get iterators to the beginning and end of the container.

- **Access via search**

`find` returns any matching element to you, or you get `end()` back if there is none. For the `multi_...` variants, the `count` and `equal_range` methods make sense: With `count`, you can count the number of matching elements, and with `contains`, you can check for existence. The `equal_range` method returns the range of all matching elements with a pair of iterators.

Here is an example of accessing an `unordered_multiset`:

```
// https://godbolt.org/z/a8cGxr9s9
#include <unordered_set> // unordered_multiset
#include <iostream>
#include <string>
using std::unordered_multiset; using std::cout; using std::string;
int main() {
    const string in = "She sells sea shells by the seashore.";
    unordered_multiset<int> cs(in.begin(), in.end()); // string as container
    cout << cs.count( 's' ) << " times s\n"; // Output: 7 times s
}
```

Listing 24.87 For the “multi” variants, “count” makes perfect sense.

If creating a key is expensive and you need to do it often for searching, consider the `ordered_multiset`. This allows you to search without creating a complete key.

Deleting

The relative order of nondeleted elements to each other remains valid when deleting, as do iterators referencing these elements:

- **Deleting all elements**

`clear()` deletes all elements.

- **Deleting or transferring a single element**

`erase` with a value deletes *all* occurrences of that element. The return value is the number of elements deleted. If you specify an iterator, the element at that position is deleted. The return value is then the position of the next element. With `extract`, you delete an element that you would normally `insert` somewhere else.

■ Deleting multiple elements

With a pair of iterators, you can delete multiple elements. This actually only makes sense for unordered containers if this range includes all elements, which is equivalent to `clear()`, or if the range includes all or some elements of an `equal_range`. Both are possible, but probably rather rare in practice. Deletion also works with areas that include several different elements, but which elements lie in between is hard to predict for unordered containers.

Specialty: By the Bucketful

You can gain detailed insights into the inner workings of the `unordered_multiset` using `bucket_count()`, `bucket_size`, and `bucket` as well as the `int` variants of `begin(int)` and `end(int)`.

Specialty: Reserve Space

To repel unwarranted interruptions from rude reallocations of the hash table when inserting, you can manipulate the hash mechanism using the `load_factor`, `max_load_factor`, `rehash`, and `reserve` methods.

24.8.7 “unordered_multimap”

The `unordered_multimap` can do everything that `multimap` can do ([Section 24.7.6](#)), but as an unordered associative container, it also has an interface that is closely related to the `unordered_multiset` ([Section 24.8.6](#)).

Keys are mapped to target values, which usually takes $O(1)$ time, but in exceptional cases can also take $O(n)$. Instead of using less than `<`, the keys are compared using `==` and a hash function.

A key can be inserted multiple times, each time referring to different target values.

Element Properties

As with all map-like containers, the elements of the `unordered_multimap` are of type `pair<const Key, Target>`. This means that if you have a reference to an element or a `Target`, you can assign a new value to `Target`. The `Key` is `const` and must not be changed under any circumstances; otherwise, the container will get confused.⁶

For the key, equality `==` must be checkable and a hash must be computable. This can be done either via the free function `operator==` and the functor `std::hash<Key>`, or you can specify your own functors as template parameters. Recall that instead of `operator==`, you can simply define `operator<=>` with `=` default if the compiler's implementation supports it.

⁶ Under no circumstances should you change a key in such a way that the change affects `==` or the hash function.

Initialize

The template parameters for defining an `unordered_multimap` are at least `Key` and `Target`. The compiler cannot spare you the template parameters when you use nested initializer lists for creation; it can only do so if you use `pair`.

Optionally, you can also specify `Hash`, `KeyEqual`, and `Allocator`.

The usual suspects are available as constructor overloads:

- The default constructor for an empty container
- A range from another constructor or stream using an iterator pair
- An initializer list with the desired values of the container
- From C++23, *from_range* and a range
- The copy and move constructors for taking over the contents of another `unordered_multimap`

The first three overloads each accept the initial size of the hash table as an optional parameter.

Assignment

You can also assign the content of another already initialized `unordered_multimap` afterward. This copies or moves the contents. `swap` is also available to you.

Insert

You insert similarly as with `multimap`. This also means that `[]` or `at` is not available to you:

- **Inserting a single element with automatic positioning**
The `insert` method takes a `pair`; `emplace` takes the two elements of the pair. Both always insert a new element as duplicate keys are also stored.
- **Inserting a single element with a position hint**
`insert` also accepts an additional iterator as a parameter; `emplace_hint` does as well. The iterator should then point to an element whose key `==` the new key, as the container can possibly save on hashing and searching work. If the hint does not fit, the advantage is lost, but the insertion is still correct.
- **Inserting multiple elements**
With an area as a parameter—that is, two iterators—or an initializer list, you can also use `insert` to insert multiple elements at once. Starting from C++23, there is also *`insert_range`*. With *`merge`*, you can retrieve all elements from another suitable container even if they are then contained multiple times here.

Iterators lose their validity with each insert operation, unless you have previously ensured enough space in the hash table using `reserve`.

Accessing

With forward iterators, not much is available to you. When you dereference them, `first` contains the key and `second` contains the target value of the element:

- **Access via iterators**

`begin()` and `end()` as well as `cbegin()` and `cend()` provide you with forward iterators to the beginning and end of the container.

- **Access via search**

`find` takes a key as a parameter and returns an iterator to one of the matching elements. `count` counts the number of matching keys, and `contains` checks if a key is present. `equal_range` returns a range containing all keys that match the searched key. For all these methods, you need a fully constructed key. Unlike `multimap`, there is no shortcut with special search functions.

Deleting

You can delete elements in the following ways:

- **Deleting all elements**

With `clear()`, you remove all elements.

- **Deleting or transferring a single element**

With a key as a parameter, `erase` deletes all occurrences of that key; with an iterator as a parameter, it deletes only the corresponding element. You get the number of deleted elements or the element following the deleted one. `extract` is used to transfer an element to another container using `insert`.

- **Deleting multiple elements**

You can delete multiple elements with two iterators. This is probably of limited use with unordered associative containers, because in such a container there can be arbitrary elements between two iterators. You wouldn't know which elements you actually delete. Therefore, only when you have exactly none, one or all, elements between the two iterators, will you know which elements you're left with after the delete operation.

Iterators remain valid if their elements have not been deleted. And you can rely on the remaining elements not being rearranged. This enables use in loops.

Specialty: By the Bucketful

You can gain insights into the hash table with `bucket_count`, `bucket_size`, and `bucket`. With `begin(int)` and `end(int)`, you get iterators to the elements of a specific bucket.

Specialty: Reserve Space

With `reserve` or `rehash`, you can influence how large the hash table should become so that the table is not immediately rearranged during upcoming insert operations and

iterators remain valid. `load_factor` and `max_load_factor` are the interfaces to the table's load factor.

24.9 Container Adapters

The container adapters are not standalone containers, but rather contain one of the actual containers to hold their data. Externally, they offer methods to perform specialized tasks.

For example, you can choose what the underlying container should be for `stack`.

```
// https://godbolt.org/z/6MKMj3Pxv
#include <stack>
void run(auto data) { /* ... */ } // C++20, abbreviated function template
run(stack<int>{});                // Default: uses vector<int>
run(stack<int, vector<int>>{});    // like the default
run(stack<int, list<int>>{});     // uses list<int>
```

Listing 24.88 Adapters work with interchangeable implementations.

Container	Description
<code>stack</code>	Last in, first out
<code>queue</code>	First in, first out
<code>priority_queue</code>	Remove the largest element very quickly
<code>flat_multi[set/map]</code>	Associative container with contiguous layout

Table 24.10 Profile: The container adapters.

In this section, `Cont` or `C` stands for one of the container adapter types like `stack<int>` and `cont` for an instance of it—for example, `queue<double> cont;`. For *element type*, I use `Elem` or `E`, and for *iterator types*, I abbreviate with `It`:

■ `stack<Elem>`

You can add individual elements to a *stack* at the top and remove them from the same end—hence the term *last in, first out* (LIFO): the last element you added is the first one you get back.

■ `queue<Elem>`

A *queue* allows you to add individual elements at *one* end and remove them at the *other* end. This results in a *first in, first out* (FIFO) approach: the element that was inserted first among several is also the first one you get back.

■ `priority_queue<Elem>`

The addition of elements to a priority queue is performed without regard to their

position. However, when elements are removed, the most front-ranked element is extracted. By default, the elements are compared with `std::less<Elem>` during insertion, but the user may specify an alternative comparison operator. A typical use case is that the comparison calculates some sort of costs within an algorithm, and by that it is guaranteed that the most cost-effective element will be processed.

■ `flat_set<E>`, `flat_map<K,V>`, `flat_multiset<E>`, `flat_multimap<K,V>`

Since C++23, the flat containers are almost drop-in replacements for `set`, `map`, `multiset`, and `multimap`. They internally use a vector instead of a tree structure, which particularly speeds up iterating over the elements and minimizes memory fragmentation. But the elements are always kept sorted, which slows down insertion and removal. Especially for (many) small containers, these adapters can be a good choice. Keys and values are stored in separate containers; instead of vector, you can choose any contiguous container.

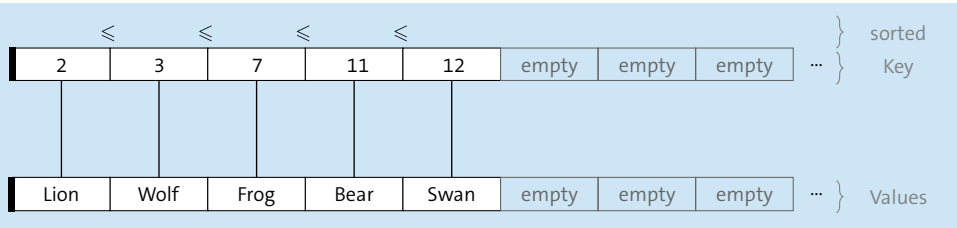


Figure 24.15 Keys and values of the flat containers are in separate contiguous containers.

Special Features of Flat Containers (C++23)

In terms of interface, `flat_set` is very similar to `set`; the same applies to the other flat containers. The keys are always kept sorted. The advantage is that the elements are stored directly next to each other in memory. The positive effects of this are that iteration is faster and memory is less fragmented (good for small elements).

Inserting and removing, on the other hand, become slower, which is particularly noticeable with many elements. How many is many? Hard to say. I have found literature that spoke of 100 elements as a lot, but also others that considered 100,000 to be small. To offer some damage control here, the constructors and methods of the flat containers that add entire ranges have an optional `sorted_unique` or `sorted_equivalent` marker parameter, which can then work more efficiently if the elements are already sorted.

Use these adapters consciously and not just always!

A special feature of the flat containers is that you can access the internal containers for keys and values with the `extract` method. The container for the keys is even sorted.

24.10 Special Cases: “string”, “basic_string”, and “vector<char>”

Container	Description
<code>basic_string</code>	Specialized container for texts, similar to <code>vector</code>
<code>string</code>	Template specialization: <code>basic_string<char></code>

Table 24.11 Profile: “string”.

Almost everything you can do with a `vector`, you can also do with a `basic_string`. The relationship goes so far that `basic_string` even has a `push_back` method for appending individual characters. And `string`, in turn, is just a template specialization of `basic_string<char>`.

Although `vector<char>` is very similar to `string`, you should use each when it semantically fits better. `string` is better suited for texts and text manipulations, while `vector<char>` is better when the characters are to be considered individually and do not necessarily form a text together. For example, some implementations have special optimizations for short strings that you wouldn't need for `vector`.⁷

However, there is at least one advantage of `string` over `vector<char>`: you can initialize a `string` directly from a string literal—that is, `const char[]`—which is not possible with a `vector<char>`. Also, `vector<char>` does not have an overloaded operator<< for output, as you can see in [Listing 24.89](#).

Since C++20, the `string` class has undergone a lot of modernization to feel more like strings from other languages. Some special features have been added that set it apart from other containers.

There are now the long-awaited `starts_with` and `ends_with` methods. Since C++23, there are also `contains`, `insert_range`, `append_range`, and `replace_with_range`. For working with APIs, `resize_and_overwrite` has been added, allowing you to overwrite a string of predefined size with a callback.

strings work excellently with ranges, just like other containers, and behave with views like containers of their underlying character types. Starting from C++23, they also have a constructor with the `from_range` tag.

Since C++17, we also have the `string_view` and, from C++20 onward, `span` extensions available:

■ `span`

A `span` roughly replaces `begin()` and `end()`. You get an extension where you can read and write. You typically use a `span` with `vector` and `array`, but it also works with `string`. You often see it used as a parameter to a function.

⁷ See “C++ Weekly—Ep 430—How Short String Optimizations Work,” https://www.youtube.com/watch?v=CIB_khrNPSU, Jason Turner, July 2 2024, [2024-05-30]

■ `string_view`

This provides a read-only span to a `string` or a `const char*`. Typically, you use it as a parameter to a function where you previously passed a `const string&`. You see an example of `string_view` as a parameter in [Listing 24.39](#).

■ `""sv` as a suffix

The standard library also provides the user-defined `""sv` literal to create a `string_view`. The advantage over `""s` for `string` is that it does not create a new `string`, but only a view on the `const char*`—so almost nothing needs to be copied or created.

```
// https://godbolt.org/z/P7zbrh71d
#include <vector>
#include <string>
#include <iostream>
#include <string_view>
using std::string; using std::string_view; using std::vector; using std::cout;

int get_len(string_view str) { return str.size(); } // string_view as parameter

int main() {
    string s1 = "Hello";           // simply with string literal
    string s2{'H','e','l','l','o'}; // or with list of char
    using namespace std::literals; // for ""s-suffix and ""sv-suffix
    auto s3 = "Hello"s;           // even simpler with real string literal
    vector<char> v1{"Hello"};      // ✗ no vector with string literal
    vector<char> v2{'H','e','l','l','o'}; // list of char is okay
    cout << s1 << s2 << s3 << '\n'; // Output of string works
    cout << v1 << v2 << '\n';      // ✗ vector has no output

    const auto str = "String"s;    // Stringliteral
    const auto strv = "String-View"sv; // String-View literal
    cout << "Length of 'str' is " << get_len(str) << '\n'; // Output: ... 6
    cout << "Length of 'strv' is " << get_len(strv) << '\n'; // Output: ... 11
}
```

Listing 24.89 “string” is more suitable for texts than “vector<char>”.

24.11 Special Cases: “vector<bool>”, “array<bool,n>”, and “bitset<n>”

For an array of truth values, the standard library offers several specialized solutions from which you can choose the right one depending on the application.

Container	Description
vector<bool>	Specialized container; stores bool compactly
array<bool,n>	Regular container; stores bool values separately
bitset<n>	Compact, but not a standard container; rather like a large unsigned with bit operations

Table 24.12 Profile: The bool containers.

24.11.1 Dynamic and Compact: “vector<bool>”

One might assume that a `vector<bool>` wastes a lot of space: a `bool` is normally as large as an `int` for performance reasons, even though a single bit would suffice to store `true` or `false`.

In the case of `vector<bool>`, the standard library has a *specialization* where bits are compacted. Multiple `true/false` values share a memory space. This means that a container `vector<bool>(100)` takes up much less memory than `bool[100]`.

Externally, `vector<bool>` behaves like a normal sequence container for `bool`—with one exception: the access methods cannot provide a `bool&` for in-place modification. A reference into the vector would always hit multiple bits at once. Instead of a `bool&`, you get back a *proxy object* that “pretends” to be a `bool&` and takes on the task of actually writing into the `vector<bool>` when it matters.

Do not use `vector<bool>` without protection against concurrent writes in parallel operation: the proxy object cannot handle it.

24.11.2 Static: “array<bool,n>” and “bitset<n>”

If you think that `array<bool,n>` is also specialized for compactness, you are mistaken. Unlike `vector<bool>`, `array<bool>` is not optimized for compactness; there is *no* specialization for it in the standard library. This means that an `array<bool,100>` takes up as much memory as `bool[100]`.

If you need a fixed-size array of `bool`, you are better off using `bitset<n>`. Although it is not a real container, it is compact and has the added advantage that you can perform bit operations directly, such as bitwise AND (&)—almost as if they were the bits in an unsigned or similar. If it is the interface of a container that you need, such as iterators, then use a `vector<bool>`. An example with some—but not all—operations is shown in [Listing 24.90](#).

```

// https://godbolt.org/z/5e3c88K8j
#include <bitset>
#include <iostream>
using std::cout;
int main() {
    std::bitset<8> bits{};           // 8 bits densely packed
    bits.set(4);                     // 5th bit to 1
    cout << bits << '\n';           // 00010000
    bits.flip();                     // invert all bits
    cout << bits << '\n';           // 11101111
    bits.set();                      // set all bits to 1
    bits.flip(1);                    // invert 2nd bit
    std::cout << bits << '\n';      // 11111101
    bits.reset();                    // set all bits to 0
    bits.set(4);                     // 5th bit to 1
    cout << bits << '\n';           // 00010000
    bits.flip();                     // invert all bits
    cout << bits << '\n';           // 11101111
    bits.set();                      // set all bits to 1
    bits.flip(1);                    // invert 2nd bit
    bits.flip(6);                    // invert 7th bit
    cout << bits << '\n';           // 10111101
    // Bitwise operations
    std::bitset<8> zack("....####", 8, '.', '#');
    cout << zack << '\n';           // 00001111
    cout << (bits & zack) << '\n'; // 00001101
    cout << (bits | zack) << '\n'; // 10111111
    cout << (bits ^ zack) << '\n'; // 10110010
    // other integer types
    std::bitset<64> b(0x123456789abcdef0LL);
    cout << b << '\n';
    // 000100100011010001010110011100010011010101110011011101110000
    cout << std::hex << b.to_ulong() << '\n'; // convert
    // 123456789abcdef0
}

```

Listing 24.90 A “bitset” example.

You can combine bitsets of the same size, similarly to how you combine integer types—at least as far as the bitwise operations `&`, `|`, `^`, `<<`, and `>>` are concerned. In addition, you have test and manipulation methods that are somewhat easier to handle than bit manipulation. In [Table 24.13](#), you can see what a `bitset` can do.

Operator	Example	Description
Constructor	bitset<8>{}	Creates a bitset with eight zeros
	bitset<8>(255)	Converts an integer to bits
	bitset<8>("1100")	Make a bitset 00001100 from a string
	...("#.",2,'.','\#')	Two characters; . false means # is true
==	as == bs	Checks for equality
!=	as != bs	Checks for inequality
[]	bs[3]	Access individual bits
test	bs.test(2)	Tests the state of a single bit
all	bs.all()	Checks if all bits are set
any	bs.any()	Checks if at least one bit is set
none	bs.none()	Checks if none of the bits are set
count	bs.count()	Counts the number of set bits
size	bs.size()	Width of the bitset
&, , ^	as & bs	Bitwise AND operation
<<, >>	bs << 1	Bitshift operators
&=, =, ^=	bs &= bs	“Immediate” bitwise operations
<<=, >>=	bs <<= 1	“Immediate” bitshift
set	bs.set(4)	Set all or one bit
reset	bs.reset(4)	Reset all or one bit
flip	bs.flip(4)	Flip all or one bit
to_string	bs.to_string()	Convert to string
to_ulong	bs.to_ulong()	Convert to unsigned long
to_ullong	bs.to_ullong()	Convert to unsigned long long
hash	hash<bitset<4>>(bs)	Support for unordered_map and the like

Table 24.13 “bitset” methods and operations.

Note on Performance

A `vector<bool>` packs multiple values into a single memory cell, which saves space. However, this approach has a downside in that changing a single value requires reading and writing back the neighboring values as well, which costs time.

A `bitset` stores each value in its own memory cell. If you want to change a single value, it can be written directly. However, this data structure requires (at least) eight times the space.

In C++, *good performance* can mean low memory usage on the one hand and low time consumption on the other—or both. Regarding memory, there is a clear statement: `vector<bool>` saves space. And speed? That is harder to answer and depends on the system. Is it faster if you don't have to consider the neighbors, or is memory plus cache so slow that using eight times more memory is also slower?

If it is important for your problem, measure and compare! I can give you a reference point. On an Intel i7-4xxx, a single experiment showed that the wasteful `bitset` is about twice as fast as the space-saving `vector<bool>`—and up to five times as fast as the naive implementation with `bool[]`.

24.12 Special Case: Value Array with “`valarray<>`”

The `valarray` is not a container like the others. Its focus is less on compatibility with iterators, algorithms, and other containers, and more on the ability to parallelize numerical data processing.

You typically use `valarray` only with numeric types because it has specialized functions for them. It is also equipped with all conceivable arithmetic operator overloads, allowing you to write entire arithmetic expressions for `valarray` as if they were numbers, similar to what mathematicians do with (mathematical) vectors and matrices.

So if you've always wished you could simply add two `vector<double>`s component-wise with `+` or multiply with `*`, then `valarray` is for you:

```
// https://godbolt.org/z/KPP16bdK9
#include <iostream>
#include <valarray>
using std::ostream; using std::valarray;
ostream& operator<<(ostream&os, const valarray<double>&vs) {
    os << "[";
    for(auto&v : vs) os << v << " ";
    return os << "]";
}
int main() {
    valarray a{ 1.0, 2.0, 3.0, 4.0 }; // valarray<double>
    valarray b{ 2.0, 4.0, 6.0, 8.0 };
```

```

valarray c{ 2.5, 1.75, 0.5, 0.125 };
valarray<double> x = ( a + b ) * c;
std::cout << "x: " << x << "\n"; // Output: [7.5 10.5 4.5 1.5 ]
auto y = ( a + b ) / 2;           // y is not necessarily a valarray!
std::cout << "y: " << y << "\n"; // Output: [1.5 3 4.5 6 ]
}

```

valarray stores its data in a one-dimensional array, but it was also designed for multi-dimensional cases. You will see examples of this later.

A particularly important feature is that valarray is intentionally free from certain forms of aliasing, allowing compilers and CPUs to optimize it very well. These certain forms allow iterators into a valarray and thus range-based for loops and similar constructs, but do not allow for permanent holding of iterators or pointers into the data.

The class is somewhat neglected in the standard due to its uniqueness. There was once an opinion that the interface of valarray was of poor quality because no one felt responsible for it. Since then, nearly two decades have passed, and apart from the support for move, the class has received small updates to work well with other new language features.⁸ Furthermore, Intel itself provides libraries that explicitly aim to accelerate valarray.⁹

Dark Corner of the Standard?

Should you use valarray or leave it alone? My advice would be to use valarray only if you want to leverage numerical computations with potential parallelization on SIMD architectures. Think again if your code is supposed to run on more than one platform (different CPUs or compilers).

So if you have the optimal numerical application and are on a type of CPU with a given compiler, then valarray might be the right choice.

Check if the *mdspan* added in C++23 is useful for you. Then you can write `cube[3, 5, 7]` because the `operator[]` also supports multidimensional indices since C++23!

With C++26, chances are that Basic Linear Algebra Subroutines (BLAS) will become part of the standard. Those would be in the `<linalg>` header and would allow you much more than matrix multiplication. They even come with parallel execution policies.

24.12.1 Element Properties

For a `valarray<T>`, `T` should be a numeric data type. I recommend a built-in integer or floating-point type. Also, `complex<>` may be suitable under certain circumstances. The

⁸ *The C++ Standard Library: A Tutorial and Reference*, Nicolai M. Josuttis, Addison-Wesley Professional 1999

⁹ *Intel's valarray Implementation*, <https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-1/intel-s-valarray-implementation.html>, [2024-08-13]

most useful `valarray` can be if `T` supports arithmetic, bitwise, and/or Boolean operations (as `int` would)—the more, the better. `T` is perfect if it supports trigonometric functions and the like, as `double` would.

Container	Description
<code>valarray<T></code>	Not compatible with other containers; for fast processing of numerical data in one or more dimensions
<code>slice</code>	Selected range of a <code>valarray</code>
<code>gslice</code>	Selected range of a <code>valarray</code>
<code>slice_array</code>	Helper class for selecting ranges of a <code>valarray</code>
<code>gslice_array</code>	Helper class for selecting ranges of a <code>valarray</code>
<code>mask_array</code>	Helper class for masking parts of a <code>valarray</code>
<code>indirect_array</code>	Helper class for a reordered view of a <code>valarray</code>

Table 24.14 Profile: Nonstandard “`valarray`” container.

There is nothing against having your own data type, except that optimizations might not apply. It may well be that you can or must use a special data type for specific hardware with a third-party library. Older GPUs, for example, only supported `float` and not `double` in their CUDA libraries, but they have since advanced. CUDA is now a widely used interface to graphics cards—more widespread than the use of `valarray`. When speed is more important than accuracy, a half-float (16-bit floating point) might be useful. Since C++23, there has been an optional built-in data type called `float16_t` (see [Chapter 4, Section 4.4.4](#)). Or you can use other libraries. You will find it under names like `float16`, `half`, `HalfFloat`, or `__fp16` and similar. CUDA now supports it, and it is used, for example, for surface textures. CUDA and `valarray` contradict each other a bit; you might have overlaps in your application.

Do Not Put Pointers into “`valarray`”

`valarray` stands for *value array*, so it is intended for values and not for any kind of indirections. Even if the implementation does not prevent this use, you should still refrain from it. Use a vector for such purposes.

Embed the examples in this section into the following template (and note my comment on `<<=` in the box that followed [Listing 24.12](#)):

```
// https://godbolt.org/z/r11584Wef
#include <valarray>
#include <iostream>
using namespace std;
```

```

template<typename T>
ostream& operator<<=(ostream &os, const valarray<T>& a) { //'<=<' with newline
    for(const auto &v : a) os << v << ' ';
    return os << '\n';
}
int main() {
    // ... example code here ...
}

```

24.12.2 Initialize

Although `valarray` is not really resizable like `vector`, you can still adjust the size of a `valarray` afterward by completely copying it. Therefore, you can initialize `valarray` without arguments or with a predefined size:

```

// https://godbolt.org/z/chnj64dva
valarray<int> data;           // initially size 0
cout << data.size() << "\n"; // Output: 0
data.resize(100);           // enlarged
cout << data.size() << "\n"; // Output: 100
valarray<int> data2(200);    // space for 200 values
cout << data2.size() << "\n"; // Output: 200
valarray<int> dataC(5, 20);  // twenty 5s, different from vector
cout << dataC.size() << ": dataC[6]=" << dataC[6] << "\n"; // Output: 20: dataC[6]=5
valarray dataD{ 2, 3, 5, 7, 11 }; // valarray<int>, initialization list
cout << dataD.size() << ": dataD[3]=" << dataD[3] << "\n"; // Output: 5: dataD[3]=7

```

As you can see, you can also predefine a number of values with initial values. But beware: the two `valarray(value, size)` parameters are in the opposite order compared to `vector`! It is better to get used to defining `valarray` only with a number for the size or an initialization list.

Copying and moving are also allowed.

Each of the `slice_array`, `gslice_array`, `mask_array`, and `indirect_array` helper classes can also be specified as a source for initialization. Note that these constructors are not marked with `explicit`, allowing implicit conversion for parameters and returns.

24.12.3 Assignment

The assignment operator `operator=` supports the same parameters as the constructor.

There is no `assign` like in many other (real) containers, but at least there is a `swap`.

24.12.4 Insert and Delete

You cannot really insert like with `insert`. However, you can access the elements for reading and writing with `operator[]`. Access is not checked; you must take care of the range boundaries yourself. There is no `at()` method for this.

However, you can still change the size of a `valarray` afterward. For this, there is the `resize` method:

- `resize(size_t size)`—enlarges or reduces, truncates or initializes with `T{}`
- `resize(size_t size, T value)`—enlarges or reduces, truncates or initializes with `value`

It is possible that all data will be copied during this operation. Pointers and iterators into the `valarray` therefore become invalid.

24.12.5 Accessing

The `operator[]` is almost magical in `valarray`. It can not only handle an integer index to address a single element, but it can also use *slices*, *masks*, and *indirections* as an index.

In fact, the aforementioned helper classes are used as an index, but in practice, many type conversions occur here, so you work with the following types as an index:

- **Slice**
You use either `slice` or `gslice`.
- **Mask**
You take a `valarray<bool>` of the same size as the data `valarray<T>`.
- **Indirection**
You take a `valarray<size_t>` of the same size as the data `valarray<T>`.

Consider the following example:

```
// https://godbolt.org/z/PeG4K7Edc
valarray v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
valarray<int> r1(v[slice(0, 4, 3)]); // Start at 0, 4 elements, step size 3
cout <<= r1;                       // Output: 14 7 10
valarray<int> r2(v[v > 6]);        // addressed by valarray<bool>
cout <<= r2;                       // Output: 7 8 9 10 11 12
const valarray<size_t> indirect{ 2, 2, 3, 6 }; // duplicates allowed
valarray<int> r5(v[indirect]);      // addressed by valarray<size_t>
cout <<= r5;                       // Output: 3 3 4 7
```

24.12.6 Specialty: Manipulate All Data

There are a handful of useful methods for information or manipulation over all data in the `valarray`:

- **sum, min, and max**

Calculate the sum, the minimum, and the maximum over all values. With a normal container, you would instead use, for example, `accumulate` or `max_element` from `<algorithm>`.

- **shift and cshift**

Shift and rotate the values around the entire `valarray`. `shift` rotates; `cshift` fills with zeros.

- **apply**

This calls a function for each element in the `valarray`. The function must take a `T` as an argument and return a `T`. You cannot modify the element. Overall, you get a new `valarray` back from `apply`. With normal containers, you would probably use `for_each`.

- **Integer, floating point, boolean, and bitwise arithmetic**

You can modify all values of a `valarray` simultaneously with all built-in operators: `+`, `*`, `+=`, `|`, `&&`, `<<`, and so on. You can specify either a single value (scalar) or, if appropriate, another `valarray` as the other operand. Writing `va * vb` is identical to `va[0]*vb[1]; va[1]*vb[1]; ...`. This applies to all binary operators.

- **Mathematical functions**

For `valarray`, a multitude of mathematical functions are overloaded, which then apply element-wise to all contained values. Examples include `abs`, `exp`, `sqrt`, `log10`, `sin`, `cos`, `asin`, and `cosh`. You receive a new `valarray` each time, as if `apply(func)` had been called.

- **Comparison operations**

You can use `operator>` and the like to compare all elements of a `valarray` against either a single value or another `valarray`. You get back a `valarray<bool>`.

24.12.7 Specialty: Slicing and Masking

The specialty compared to normal containers is that you can *slice* a `valarray` with helper classes to select only parts of it. These slices can be complicated and, for example, include a step size, thus simulating multidimensionality. With *masking*, you can exclude parts of a `valarray` from operations.

A slice helper object represents a complicated index into a `valarray`, similar to how 12 in `a[12]` would work in a normal array. You create a slice helper object with one of the constructors:

- `slice()`—empty slice
- `slice(size_t start, size_t size, size_t stride)`—parameters for start, size, and stride
- `slice(const slice& other)`—for a copy

You can then use this helper object as an index in `[]`. You get back a `valarray` or a proxy object that can be converted into a `valarray`. For example, if you have a `valarray` with values from 1 to 12, `slice(0,4,3)` addresses the values 1, 4, 7, and 10, which are exactly the values that these numbers would occupy in the first column of a 3×4 matrix.

While `slice` extracts a one-dimensional slice from a two-dimensional matrix, the `gslice` helper class is used for multidimensional slices. For example, if you consider the values 1 to 12 as a three-dimensional matrix/cuboid of size $2 \times 3 \times 2$, then `v[gslice(0, {2,3}, {6,2})]` addresses the two-dimensional slice of the front 2×3 face with the values 1, 3, 5 and 7, 9, 11:

```
// https://godbolt.org/z/ehP47vW9o
valarray<int> v {
    1,  2,  3,
    4,  5,  6,
    7,  8,  9,
    10, 11, 12 };
v[slice(0, 4, 3)] *= valarray<int>(v[slice(0, 4, 3)]); // square the first column
cout <<= v; // Output: 1 2 3 16 5 6 49 8 9 100 11 12
v[slice(0, 4, 3)] = valarray<int>{1, 4, 7, 10}; // restore
valarray<int> r3(v[gslice(0, {2, 3}, {6,2})]); // 2-D slice from the 3-D cube
cout <<= r3; // Output: 1 3 5 7 9 11
valarray<char> text("now it really starts", 20);
valarray<char> caps("NIRS", 4);
valarray<size_t> idx{ 0, 4, 7, 14 }; // Indexes in text
text[idx] = caps; // assign indirectly
cout <<= text; // Output: Now It Really Starts
```

First, the program interprets the `valarray v` as a two-dimensional matrix of size 2×3 . With a simple `slice`, I address the first column and multiply it by itself using `*=`. As a result, you see that only 1, 4, 7, and 10 were overwritten.

For clarity, I then restore the original state, as the next step interprets the 12 values as a three-dimensional object—that is, a cuboid with edge lengths $2 \times 3 \times 2$. Here, `gslice` cuts out a two-dimensional piece.

Finally, I use a `valarray<size_t>` as an argument for `operator[]` to perform an indirect assignment of another data `valarray`. `text[idx] = caps` assigns new values to the four letters addressed by `idx`.

In matrix multiplication, you might know that a matrix of size 2×4 can be multiplied by another matrix of size 4×2 ; in this product, the rows of one matrix are multiplied by the columns of the other matrix. For this, you can also use `valarray` with clever slicing:

```
// https://godbolt.org/z/hf8rqz5x5
#include <iostream>
#include <iomanip> // setw
```

```

#include <valarray>
using namespace std;

/* Print matrix */
template<class T>
void printMatrix(ostream&os, const valarray<T>& a, size_t n) {
    for(size_t i = 0; i < (n*n); ++i) {
        os << setw(3) << a[i];          // Print value
        os << ((i+1)%n ? ' ' : '\n');    // next line?
    }
}

/* Matrix cross product */
template<class T>
valarray<T> matmult(
    const valarray<T>& a, size_t arows, size_t acols,
    const valarray<T>& b, size_t brows, size_t bcols)
{
    /* Condition: acols==brows */
    valarray<T> result(arows * bcols);
    for(size_t i = 0; i < arows; ++i) {
        for(size_t j = 0; j < bcols; ++j) {
            auto row = a[slice(acols*i, acols, 1)]; // Row
            auto col = b[slice(j, brows, bcols)];   // Column
            result[i*bcols+j] = (row*col).sum();    // Cross product of row a[i] and
                                                    // column b[j]
        }
    }
    return result;
}

int main() {
    constexpr int n = 3;
    valarray ma{1,0,-1, 2,2,-3, 3,4,0};           // 3 x 3 matrix
    valarray mb{3,4,-1, 1,-3,0, -1,1,2};          // 3 x 3 matrix
    printMatrix(cout, ma, n);
    cout << "  -times-\n";
    printMatrix(cout, mb, n);
    cout << "  -results in-:\n";
    valarray<int> mc = matmult(ma, n,n, mb, n,n);
    printMatrix(cout, mc, n);
}

```


The output is the $\{\{4, 3, -3\}, \{11, -1, -8\}, \{13, 0, -3\}\}$ matrix, which represents the product of the matrix multiplication. Note that `matmult` should only receive matrices and size specifications that are compatible for this operation.

With `a[slice(...)]` and `b[slice(...)]`, a row or column is cleverly sliced out here. And as the rule of matrix multiplication states, these are then multiplied element-wise and summed up: this is done with `valarray` in the innocuous-looking expression `(row * col).sum()`.

I deliberately used the compiler's type inference with `auto` for `row` and `col`, instead of writing `valarray<int>` for the row and column. It is possible that the compiler optimizes here with a special object.

The program's output is this:

```
1   0  -1
2   2  -3
3   4   0
-times-
3   4  -1
1  -3   0
-1   1   2
-results in-:
4   3  -3
11  -1  -8
13   0  -3
```

`valarray` is not very common. However, you can find more examples in *Thinking in C++*.¹⁰

¹⁰ *Thinking in C++ Vol 2 - Practical Programming*, http://www.linuxtopia.org/online_books/programming_books/c++_practical_programming/c++_practical_programming_202.html, [2017-03-11]

Chapter 25

Container Support

Chapter Telegram

- **Iterator**

A model for referencing elements in containers.

- **Algorithmus**

An advanced operation on the contents of a container.

- **Range**

An advanced model related to iterators. A range represents a sequence that can stand alone but usually refers to a container.

- **View**

A lightweight range that usually refers to a range or a part of it. While ranges exist as a concept, views are concrete instances that represent a combination of a range and an operation or a filter. Views are *lazy* and can be combined into pipelines with the operator `|`.

- **Range adapter**

A function object that creates views; that is, it defines an operation or a filter on a range.

- **Range factory**

A function object that is not based on a range but generates values as a range itself.

Algorithms in the standard library are utility functions that operate on iterators, usually in such a way that the iterators define *areas* into containers to which the algorithm is applied. So you will often see that parameters come in pairs of iterators—namely, `begin` and `end`.

Ranges are the successors of these iterator pairs and, together with views, they extend the algorithms. All algorithms from `std` that take pairs of iterators can now also be found in `std::ranges` with ranges as parameters—the standard then calls them *constrained algorithms*. I find the name misleading because the algorithms in `std::ranges` are not restrictive at all, but rather extended:

- They accept ranges as parameters but also work with pairs of iterators.
- They typically return ranges, so they can be directly combined with views.

- They consistently support an optional projection parameter, which allows access to the range elements using a functor—for example, by specifying a member function.

You can therefore expect that range algorithms will be extended sooner than classical algorithms. This is already the case. For example, `ranges::starts_with()` no longer has a counterpart in `std`. I mentioned in the previous chapter on containers what algorithms and ranges are because they go hand in hand. Algorithms, ranges, and views from the `<algorithm>`, `<numeric>`, and `<ranges>` headers mostly work with containers. You can also write your own algorithms and views.

In this chapter, I will try to familiarize you with algorithms, ranges, and views. When I speak of an *algorithm*, I usually mean functions from `<algorithm>` in the `std` namespace—the classical ones that take iterators as parameters to describe an area inside a container. Views and ranges are new in C++20, come from the `<ranges>` header, and reside in the `std::ranges` and `std::ranges::views` namespaces. In particular, views often implement algorithms as well, but in the sense of the standard library, they are not considered algorithms. However, they also support working with containers.

Namespaces for Algorithms, Ranges, and Views

For ranges, the `std::ranges` namespace was introduced. It is quite common to abbreviate it with `using rng = std::ranges` (or with `rg`). The namespace `std::ranges::views` contains the views. It has an alias, `std::views`, and is often abbreviated with `using vws = std::ranges::views` (or `vs`).

Some things are defined in both `std` and `std::ranges`. This is the case, for example, with `size()`, `begin()`, and `sort()`. You should then prefer the new namespace, as the new variants often include additional functions and also address some defects.

Do not use `using namespace std::ranges` and `using namespace std::views`! Things can get messed up and be a source of nasty bugs. In this book, I usually use the `ranges`, `rng`, and `rs` aliases as well as `views` and `vs`.

25.1 Algorithms

What is an *algorithm* in the context of the standard library? The classical algorithms are each a *function template*. In a broader sense, the new views are also algorithms. This function or class does something *useful*, and what it does, it does in a very *general*—generic—way. The fact that the algorithm functions and views are actually templates is precisely because they are meant to work generically: they should work on as many sequences as possible with as many element types as possible.

Every algorithm must be well-documented in its behavior. You need to be able to read the documentation and know exactly how the algorithm behaves when working with

your data. And if it does something harmful, such as consuming too much memory or running too long, this case must also be described in the documentation.

If you need to solve a problem that is already solved or can be easily solved with `<algorithm>` or `<numeric>` or `<ranges>`, then you should use them:

- The functions and classes are tested by many programmers and are continuously maintained.
- The algorithms and range adapters are building blocks that you can assemble into complex tasks.
- It is therefore easier to write code with building blocks.
- Code made from building blocks is easier to debug.
- Code made from modules is especially easier to verify later and better to revise.

I would like to illustrate the last three points with an example.

```
// https://godbolt.org/z/jcWvThn3a
std::vector v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for(size_t i=1; (i < v.size()) && flag; ++i) {
    flag = false;
    for(size_t j=0; (j < v.size()-i); ++j) {
        if(v[j+1] < v[j]) {
            std::swap(v[j+1], v[j]);
            flag = true;
        }
    }
}
for(int i:v) std::cout << i << ' ';
```

Listing 25.1 Something is done with the input vector—but what?

```
// https://godbolt.org/z/EjT7nPPv7
std::vector v{0,1,3,5,7,9,2,4,6,8};
std::sort(v.begin(), v.end());
for(int i:v) std::cout << i << ' ';
```

std::cout << '\n';

// or with a range:

```
std::ranges::sort(v);
for(int i:v) std::cout << i << ' ';
```

std::cout << '\n';

Listing 25.2 This input vector will be sorted!

If you look at [Listing 25.1](#), you might eventually say: “That probably implements Bubblesort.” But for [Listing 25.2](#), you can immediately say: “That sorts.” And that’s not all:

How long does it take for you to be able to say that both listings are *correct*? Why does [Listing 25.1](#) sometimes have in the loop `i=1; (i < v.size())` and other times `j=0; (j < v.size()-i)`? What happens with an empty vector or one with only a single element? Let me tell you: the listing is correct, and—yes—it actually implements Bubblesort. But you surely understand what I mean. It took either more time (or/and experience) to gain this certainty with [Listing 25.1](#), or my assurance that it is a correct Bubblesort.

25.2 Iterators and Ranges

A detailed introduction to iterators can be found in [Chapter 24, Section 24.2](#). How ranges and views work is detailed in [Chapter 24, Section 24.1.7](#). Here I will summarize the essential things that are important for working with algorithms.

Every algorithm and every range adapter imposes certain requirements on the iterators or ranges it receives as parameters. Before C++20, iterator categories were important; now they have been replaced by actual concepts. Categories and iterator concepts build on each other. The rule of thumb is this: the higher the category, the more requirements for the iterator; it must support more operations.

The iterators of each container type are associated with a corresponding category. This allows you to see which containers are compatible with an algorithm. Don't forget that streams also offer iterators and can therefore work with many algorithms.

I will show you the hierarchy of iterator categories. In C++20, the names of the categories have been given the prefix *Legacy*, which I will omit here. If an algorithm requires, for example, that the iterators `begin` and `end` follow the *ForwardIterator* category, it means there may be multiple assignments `it = begin`, increment with `++it`, and read with `*it`:

- *Iterator*: only increment required; only traversed once
- *OutputIterator*: like *Iterator* and must support writing
- *InputIterator*: like *Iterator* and must support reading
- *ForwardIterator*: like *InputIterator* and must also be traversable multiple times
- *BidirectionalIterator*: like *ForwardIterator* and must be traversable backward
- *RandomAccessIterator*: like *BidirectionalIterator* and also random access
- *ContiguousIterator*: like *RandomAccessIterator* and also the elements must be contiguous in memory

Since C++20, these have been replaced by actual concepts. These also build on each other, but there are about twice as many concepts as for iterators. The most important ones correspond to the categories. For example, there is an `input_iterator` concept, which corresponds to the *InputIterator* category. I will spare you an explicit listing.

Much more important are the range concepts. These also align with the iterator categories. So there is—you guessed it—a `ranges::input_range`, which states that the range is built with an `input_iterator`. It's that simple. Almost. Because there are a handful of other range concepts that go beyond the iterator categories:

- **view**

The range is a view, so unlike containers, it can be copied efficiently.

- **borrowed_range**

The range provides iterators that remain valid even if the range becomes invalid. Note that the container to which these iterators refer must still be valid.

- **sized_range**

Can specify the number of elements.

- **common_range**

The sentinel is an iterator, `begin()` and `end()` thus have the same type. This is true for all containers and many views, but not, for example, for the `take_view`, `drop_view`, or `iota_view` view.

- **viewable_range**

Can be converted into a view with `views::all`.

- **constant_range**

The elements are not modified, similar to `const_iterator`.

When I list the algorithms later, I use the terms from [Table 25.1](#). You can see which iterator requirement demands which operations. The table also provides examples of containers that meet the requirements. For example, you can see that the requirements for bidirectional iterators are met by `vec`, `list`, and `set`, but not by `unordered_set`. I have abbreviated `vector` as `vec` and `unordered_set` as `u-set` and mentioned both only as examples.

Requirement	Abbr.	Possible Parameters	vec	list	set	u set
Input	in	in, fw, bi, ra, ct	Yes	Yes	Yes	Yes
Output	out	out, fw, bi, ra, ct	Yes	Yes	Yes	Yes
Forward	fw	fw, bi, ra, ct	Yes	Yes	Yes	Yes
Bidirectional	bi	bi, ra, ct	Yes	Yes	Yes	—
Random access	ra	ra, ct	Yes	—	—	—
Contiguous	ct	ct	Yes	—	—	—

Table 25.1 Iterator requirements and their compatibility with example containers.

25.3 Iterator Adapter

Iterators are a central part of the standard library and primarily a concept. Through further abstractions, the behavior of algorithms can be modified surprisingly often using *iterator adapters* without disturbing the idea. Many of these adapters can be found in `<iterator>`. I cannot cover all possible combinations in the explanatory list about algorithms, but to give your imagination some stimulus, here are some examples:

- With a `move_iterator`, you can convert all copy operations of an algorithm into move operations.
- With `back_inserter`, you do not need a preexisting target range; instead, each copy is converted into a `push_back`.
- The same is done by `front_inserter`, but it calls `push_front`.
- Even more flexible is `inserter`, which successively calls `insert` at a specific position.
- An `ostream_iterator` makes a copy to an output in a stream.
- The same is done by `istream_iterator`, but it copies values from a stream into a container.
- `reverse_iterator` converts every `++` operation into a `--`.

25.4 Algorithms of the Standard Library

All algorithms, ranges, and views in the context of this chapter are function templates or classes from the `<algorithm>` and `<ranges>` headers. Some come from `<numeric>`, which I will mention separately. Typically, algorithms work together with *containers* and use *iterators* or *ranges* as means. This, in turn, results in algorithms also working with *streams*, as they also offer iterators.

The fact that all algorithms are implemented as templates means that the code of the algorithm can automatically be used by the compiler as `inline` on the spot. Unlike with object hierarchies with potentially virtual methods that prevent inlining, this provides as much performance as one can possibly get. For the algorithms of the standard library, maximum speed within the scope of their general implementation is one of the top design principles.

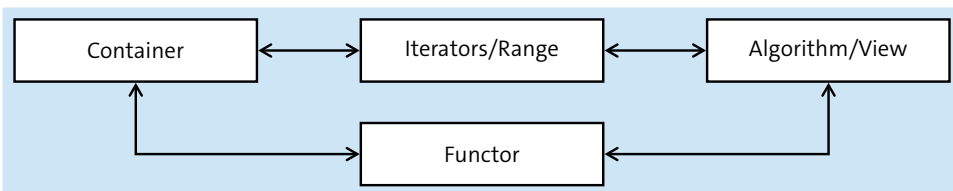


Figure 25.1 The containers do not work directly with the algorithms but “communicate” with them through iterators.

Algorithms Do Not Delete or Insert

Algorithms and range adapters consistently work only on existing ranges. If they produce an output, the target range must already exist. Algorithms *never insert themselves*. However, you can achieve this with an iterator adapter.

Also, *removed algorithms and range adapters never actually delete themselves*. They simulate this by rearranging elements and inform you where the unused elements are located. You, as the user, are responsible for the actual removal. Algorithms cannot assume to know whether and how elements are truly removed or destroyed. The most obvious representative of this group is `remove` (and `ranges::remove`), but also `unique` (and `ranges::unique`) only moves the “removed” elements to the end of the specified range. The return value of these functions is then an iterator `it`. You will actually remove elements, for example, with `cont.erase(it, cont.end())`.

The functions are always very generic and very abstract. Sometimes you have to think a bit outside the box to figure out how to combine algorithms for a complex task. An example: Do you know the `cat data.txt | sort | uniq` Unix command sequence? This removes all duplicate lines from the file `data.txt`. The same can be done with C++ algorithms if you combine them in the right way.

```
// https://godbolt.org/z/9xrP8Ehvd
#include <algorithm>
#include <vector>
#include <iostream>
void sort_uniq(std::vector<int> &data) {
    std::ranges::sort(data);           // sorting
    auto to_delete = std::ranges::unique(data); // move to the back
    data.erase(to_delete.begin(), to_delete.end()); // actually delete
}
int main() {
    std::vector ns{1,5,2,3,9,2,2,2,2,1,5,2,2,3,1,1,2,2,1};
    sort_uniq(ns);
    std::ranges::for_each(ns, [](auto x) {
        std::cout << x << ' '; });
    std::cout << '\n'; // Output: 1 2 3 5 9
}
```

Listing 25.3 Composing functions.

Here I combine the `sort` and `unique` functions with the `erase` method of `vector` to replicate the Unix `sort | uniq`. The sorted result contains each element only once.

Only `sort` and `unique` are algorithm functions. To actually delete, we need a container method—and in the case of `vector`, it's `erase`. Because this method exists for almost all containers, a generalization from `vector<int>` to other standard containers would look identical. However, note that to call a container *method* in your own function like `sort_uniq`, you must pass the container as an argument. In a true algorithm function, this is never the case; they always use *iterators* or *ranges* to operate on a portion of any container or stream.

At the end of this chapter, you will find tips for developing your own algorithms. The fact that the interfaces are ranges and not containers is the most important rule.

25.5 Parallel Execution

All algorithms are executed sequentially unless you are programming in C++17 and use an *execution policy* from the `<execution>` header as an additional first parameter to an algorithm function template. Not all algorithms support all policies:

- **`std::execution::seq`**
The algorithm is not parallelized and runs in the calling thread. However, unlike without this parameter, the steps can be executed in a different order.
- **`std::execution::par`**
The steps can be executed in parallel. The work is divided among multiple threads, and each thread executes individual steps sequentially in an unspecified order.
- **`std::execution::unseq_par`**
The steps can be executed in parallel and simultaneously. In addition to `par`, each thread can execute multiple steps at once here—that is, they can *vectorize*. Special CPU registers and instructions (MMX, SSE, AVX) are often used.
- **`std::execution::unseq`**
Starting with C++20, you can request `unseq` vectorization without multithreading. This makes debugging particularly easier.

Especially with `unseq_par`, it could really take off—if the compiler, the standard library used, the element types, the algorithm employed, and the CPU or additional hardware all fit together optimally. I translated [Listing 25.4](#) with GCC 13.2, installed Intel oneAPI Threading Building Blocks (<https://askubuntu.com/a/1183811>), and used them with `-ltbb`.

Note that with the Compiler Explorer URL (godbolt.org) in the listing, I added the extra library in the interface. I needed to use GCC 11.2 though, a later version did not work with that specific installation. Locally I used GCC 13.2 with the library installed in docker. This is my Dockerfile `gcc13.Dockerfile`:

```
FROM gcc:13.2
RUN apt-get --allow-unauthenticated --allow-insecure-repositories update && \
    apt-get install --allow-unauthenticated -y libtbb-dev && \
    rm -rf /var/lib/apt/lists/*
CMD [ "/bin/bash" ]
```

This is how you build the gcc13 Docker container:

```
docker build -f gcc13.Dockerfile -t gcc13 .
```

You can now compile in this container:

```
docker run --rm -i -t --volume $(pwd):/workdir --workdir /workdir gcc13 \
    g++ -O3 -std=c++23 -ltbb 25AlgoPar.cpp -o GCC-25AlgoPar.x
```

You also use it to execute, as in the next listing.

```
docker run --rm -i -t --volume $(pwd):/workdir --workdir /workdir gcc13 \
    ./GCC-25AlgoPar.x
// https://godbolt.org/z/Koxoha5bx
#include <algorithm> // find
#include <numeric>    // reduce, accumulate
#include <execution>  // std::execution
#include <iostream>
#include <chrono>     // time measurement
using namespace std::chrono;
long long millisSince(steady_clock::time_point start) {
    return duration_cast<milliseconds>(steady_clock::now()-start).count();
}
template <typename FUNC> void timeit(const char* title, FUNC func) {
    auto start = steady_clock::now();
    auto ret = func(); //execute
    std::cout << title << ": " << millisSince(start) << " ms" << std::endl;
}
int main() {
    using namespace std::execution; // seq, par, par_unseq
    using std::reduce; using std::accumulate; using std::find;
    std::vector<double> v(600'000'000, 0.0); // 600 million elements
    for(auto&x:v) x = ::rand(); // fill with random values
    timeit("warm-up", [&v] {
        return reduce(seq, v.begin(), v.end(), 0.0);
    });
    timeit("accumulate", [&v] {
        return accumulate(v.begin(), v.end(), 0.0);
    });
```

```
    timeit("reduce, seq      ", [&v] {  
        return reduce(seq, v.begin(), v.end(), 0.0);  
    });  
    timeit("reduce, par      ", [&v] {  
        return reduce(par, v.begin(), v.end(), 0.0);  
    });  
  
    timeit("reduce, par_unseq", [&v] {  
        return reduce(par_unseq, v.begin(), v.end(), 0.0);  
    });  
    timeit("find, seq        ", [&v] {  
        return find(seq, v.begin(), v.end(), 1.1) == v.end() ? 0.0 : 1.0;  
    });  
    timeit("find, par        ", [&v] {  
        return find(par, v.begin(), v.end(), 1.1) == v.end() ? 0.0 : 1.0;  
    });  
    return 0;  
}
```

Listing 25.4 Algorithms run in parallel.

The `accumulate` and `reduce` algorithms compute the sum of a container. The latter is designed for parallelism; while `accumulate` has a fixed grouping, `reduce` does not. For example, if `plus +` is the operation and `1,2,3,4,5,6` are the data, then `accumulate` guarantees $(((((1+2)+3)+4)+5)+6)$. With `reduce`, however, it can be $(((((1+2)+3)+4)+5)+6)$, $(1+(2+(3+(4+(5+6))))))$, $((1+2)+(3+4)+(5+6))$, or any other grouping. This is much better suited for parallelization, but it is not possible with all operations. With `minus -`, you cannot perform such a reordering without changing the result.

Thus, `reduce` is a variant provided specifically for parallelizability, similar to `accumulate`. The same applies to `inclusive_scan` and `exclusive_scan` for `partial_sum` as well as `transform_reduce` for `inner_product`.

Note that some algorithms have few requirements for iterators—for example, only `input_iterator` for `copy` or `reduce`. When you specify an execution policy, it usually changes to the somewhat stricter `forward_iterator`. I won't mention this in the later lists.

The output of the program on my eight-core processor is as follows:

```
warm-up          : 431 ms  
accumulate       : 899 ms  
reduce, seq      : 450 ms  
reduce, par      : 111 ms  
reduce, par_unseq: 111 ms  
find, seq        : 283 ms  
find, par        : 110 ms
```

As you can see, the parallel algorithms run much faster with `par` here. Intel oneAPI Threading Building Blocks allows vectorization in the special CPU registers with `par_unseq` and can potentially speed up further, but not in this case.

A compiler implementation or library can offer additional policies—for example, for execution on specific devices like the graphics card.

In all modes, the function you execute must be free of dependencies on other steps. The algorithms themselves have no synchronization. If the steps in `par` use shared resources, you must handle synchronization yourself. In `unseq_par` and `unseq`, synchronization is not allowed because it doesn't make sense with vectorization.

25.6 Lists of Algorithm Functions and Range Adapters

In this section, you will find an overview of all the algorithms in the standard library. Some you will surely use more frequently, while others are only used in very specific cases. On the other hand, to even know that you could use an algorithm, you should get an overview.

In the lists, I have added a frequency column. This represents, completely subjectively, how likely it is that you will use the corresponding function:

- **Constant**

You should definitely memorize these functions, as they can almost always be useful when working with streams or containers.

- **Frequent**

Once you get a feel for algorithms, you will surely soon start using these functions.

- **Occasional**

For these functions, you already need a special task.

- **Rare**

Functions marked like this will only be used for very specific problems. However, they solve complex tasks in such a clever way that their use is worthwhile when it fits.

- **Special**

The use of the few functions marked this way requires the deepest expert knowledge and has extreme conditions for use. I have marked the functions for uninitialized memory this way.

However, it would be completely wrong to ignore those functions whose names do not immediately mean something to you or seem too complicated. Almost all algorithms can be useful in everyday programming. Therefore, I recommend taking a close look at the actual list; prioritizing is fine. But always return to the list, look around the neighborhood, and occasionally think outside the box—especially abstractly. You will be amazed at how useful algorithms with such quirky names as `remove_copy_if` can be.

In addition, I have specified for each function which category of iterators or concepts of ranges the algorithm requires. Some algorithms take two (or more) groups of iterators or ranges, for example, different ones for input and output. Algorithms that require contiguous ranges do not yet exist. I introduce some abbreviations for them here to refer to them in later in the list of algorithms:

- `in` (`input_range`) for input, forward, bidirectional, or random-access ranges or iterators
- `out` (`output_range`) for output, forward, bidirectional, or random-access ranges or iterators
- `fw` (`forward_range`) for forward, bidirectional, or random-access ranges or iterators
- `bi` (`bidirectional_range`) for bidirectional or random-access ranges or iterators
- `ra` (`random_access_range`) for random-access ranges or iterators

In [Table 25.1](#) of the previous section, you will find examples of containers that are compatible with each.

This is what an example entry in the lists looks like:

- **`copy`**
Usage: constant; iterators: in/out
Copies a range element-wise into a target range.
- **`ranges::starts_with`**
Usage: frequent (C++23); iterators: fw
Checks if one range starts with another.

The function of the algorithm is called `copy`. Its full name is classically `std::copy` and the new `std::ranges::copy`. Sometimes there is no classic variant anymore, as is the case with `starts_with`; then I explicitly prefix it with `ranges::`.... Next is the frequency of use of the function. As a function in the constant category, you should definitely have `copy` in your repertoire. If it is a very new function, I mention that with “C++23”. Rightmost, I note the requirements for the iterators or ranges. The `copy` function requires input iterators for some of its parameters and output iterators for other parameters. This is indicated in the list with “Iterators: in/out”. The function `starts_with` is satisfied with forward iterators, which is indicated with “fw”. Based on [Table 25.1](#) or the description of the individual containers, you know that almost all containers work with `copy`. Below that line, you see a brief description of the algorithm.

25.6.1 Range Adapters and Views

And now we come to the exceptions. The range adapters are defined in the header `<ranges>`. They *support* the algorithm functions or can replace them through clever combinations. They are also not functions but classes. I start with the range adapters because they might represent the future. They were introduced in C++20 and received

a major update in C++23, and there are already plans for what will be added in C++26. I particularly highlight things that were added only in C++23.

Range adapters are function objects in `std::view` that create instances of a template class from `std::ranges`. For example, `std::views::take()`, when combined with a range, creates an instance of type `std::ranges::take_view`—the actual view. Thus, you can usually create the view directly with the type or with a range adapter—typically with the same name without `_view`, but there can be other ways (which you might write yourself).

```
// https://godbolt.org/z/rrq1v5qPs
std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto take5 = rs::take_view{lst, 5};    // View via type
auto take6 = lst | vs::take(6);        // View via adapter
```

Listing 25.5 View types and their adapters.

All views implement the interface `std::ranges::view_interface` and thus have the following member functions, some of them only under certain conditions:

- **empty(), operator bool() for if(v)**
Check if a `forward_range` or `sized_range` is empty.
- **size()**
Returns the number of elements in a `sized_sentinel_for` view.
- **front(), back()**
Returns the first element for `forward_range` or the last element for `bidirectional_range`.
- **operator[n]**
Is the *n*-th element of a `random_access_range`.
- **data()**
Returns a pointer to the data of a `contiguous_range`.
- **cbegin(), cend()**
From C++23, they return constant versions of `begin()` and `end()`.

One of the most important properties of views is that they are *lazy*. For example, if you execute `auto v = mydata | take(5)`, initially *nothing* happens! The type of `v` is a *view* and contains no data. Only when you iterate over `v` and do something with the elements is the operation executed and the result element produced. Typically, you materialize a view with `ranges::to()`, the *from_range* constructors of the containers, or in a loop.

- **views::all**
Usage: rare; iterators: –
All elements of a range.
- **ranges::subrange**
Usage: frequent; iterators: in/out

Creates a view from an iterator and a sentinel. Some adapters, factories, and algorithms also use this type. Useful: When an algorithm returns a subrange, you can extract the iterators of the range with `auto [beg, end] = ...`

- **`views::empty`**
Usage: occasional; iterators: –
Creates an empty view with no elements.
- **`views::single`**
Usage: occasional; iterators: –
Produces a view that contains a single element.
- **`views::iota`**
Usage: occasional; iterators: –
For an ascending sequence of values. Corresponds to the `iota` algorithm.
- **`views::istream`**
Usage: rare; iterators: in
A view created by multiple `operator>>`.
- **`views::repeat`**
Usage: occasional (C++23); iterators: in
Repeats the same value.
- **`views::filter`**
Usage: frequent; iterators: in
Filters out elements from a range that satisfy a predicate. Note that you may only modify the result values if the modified value also satisfies the predicate!
- **`views::transform`**
Usage: frequent; iterators: in
Applies a function to each element.
- **`views::counted`**
Usage: rare; iterators: in
Creates a range from an iterator and a count *n*.
- **`views::take`**
Usage: frequent; iterators: in
Passes only the first *n* elements of a range.
- **`views::take_while`**
Usage: frequent; iterators: in
Passes elements as long as a predicate is true.
- **`views::drop`**
Usage: frequent; iterators: in
Skips the first *n* elements of a range.
- **`views::drop_while`**
Usage: frequent; iterators: in
Skips elements as long as a predicate is true.

- **views::join**
Usage: frequent; iterators: in
Turns a range-of-ranges into a flattened range.
- **views::join_with**
Usage: frequent (C++23); iterators: in
Like `ranges::join_view`, but with a separator element between the joined ranges.
- **views::split**
Usage: frequent; iterators: fw
A `forward_range` is split into multiple ranges at a delimiter.
- **views::lazy_split**
Usage: rare; iterators: in
Also splits, but only requires an `input_range`.
- **views::reverse**
Usage: frequent; iterators: in
Reverses a range.
- **views::enumerate**
Usage: frequent (C++23); iterators: in
Turns a range of elements into a new range of tuples, each containing an index and the element.
- **views::elements**
Usage: occasional; iterators: in
Turns a range of tuples into a range of the n -th elements.
- **views::keys**
Usage: occasional; iterators: in
Like `elements` for the first tuple element.
- **views::values**
Usage: occasional; iterators: in
Like `elements` for the second tuple element.
- **views::zip**
Usage: constant (C++23); iterators: in
Takes multiple n ranges and creates a range of tuples with n elements each.
- **views::zip_transform**
Usage: frequent (C++23); iterators: in
Like `zip`, but then calls a function with n arguments.
- **views::chunk**
Usage: frequent (C++23); iterators: in
Splits a range into multiple ranges of length n , without overlap.
- **views::chunk_by**
Usage: frequent (C++23); iterators: in
Groups consecutive elements based on a criterion. Reminiscent of SQL's `group by`.

- **`views::slide`**
Usage: rare (C++23); iterators: fw
Splits a range into multiple ranges of length n , with overlap.
- **`views::stride`**
Usage: occasional (C++23); iterators: in
Takes every n -th element of a range.
- **`views::adjacent<N>`**
Usage: rare (C++23); iterators: fw
Creates a range of tuples, each containing N adjacent elements of the input range.
Unlike *slide*, N is a template parameter.
- **`views::pairwise`**
Usage: rare (C++23); iterators: fw
A specialization of *adjacent<2>*.
- **`views::adjacent_transform<N>`**
Usage: rare (C++23); iterators: fw
Grouped like *adjacent<N>* (or *slide*), but then calls a function with N arguments.
- **`views::pairwise_transform`**
Usage: rare (C++23); iterators: fw
A specialization of *adjacent_transform<2>*.
- **`views::cartesian_product`**
Usage: occasional (C++23); iterators: in/fw
Generates all combinations of elements from multiple ranges.

There are several other classes that help you especially when designing your own adapters:

- **`ranges::view_interface`**
A base class for custom views.
- **`ranges::range_adaptor_closure (C++23)`**
A base class for custom view adapters for the operator|.
- **`ranges::ref_view`**
All elements of a range as references.
- **`ranges::owning_view`**
Needed to obtain the values of a temp-value view via *move*. You can use the *all* adapter to generate it. This is (so far) the only view that contains elements itself. It is still cost-effective, as it only allows movable temp-values as parameters.
- **`views::common`**
Transforms a view with a sentinel into a view with an *end()* iterator, making it behave more like a container.
- **`views::as_const (C++23)`**
Protects the *elements* of a view by converting it into a *constant_range*, making the

elements immutable. This is different from how `std::as_const` works or how a `const` before a view variable affects it, which unfortunately leaves the elements mutable.

■ **`views::as_rvalue` (C++23)**

A view of a sequence that converts each element into an rvalue.

You need to be aware that views have different performance characteristics depending on the type of range (or underlying container) you use them on. Consider, for example, that `drop(10)` on a vector is practically instantaneous with $O(1)$, but on a `list` it is linear with $O(n)$ and therefore much slower. Preceding views also have an impact. The standard library tries to choose performant implementations and sometimes caches intermediate results. For instance, `begin()` after a `filter` is always slow because the elements to be filtered at the beginning must be traversed. However, this would be very disadvantageous for something as important as `begin()`. Therefore, `filter` caches the result of `begin()` and returns it on each subsequent call.

Table 25.2 provides a brief overview of how some views perform on different ranges, so you can get a sense of what to watch out for. For example, you can see that `size(...)` is no longer possible after a `filter`.

Sequence	1st <code>begin()</code>	2nd <code>begin()</code>	<code>size()</code>	1st <code>empty()</code>	2nd <code>empty()</code>
vector <code>vec</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
list <code>lst</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>vec drop</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>lst drop</code>	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>vec filter</code>	$O(n)$	$O(1)$	–	$O(n)$	$O(1)$
<code>lst filter</code>	$O(n)$	$O(1)$	–	$O(n)$	$O(1)$
<code>vec filter drop</code>	$O(n)$	$O(1)$	–	$O(n)$	$O(1)$
<code>lst filter drop</code>	$O(n)$	$O(1)$	–	$O(n)$	$O(1)$

Table 25.2 Performance of various container and view combinations.

The innovations brought by C++23 are particularly interesting. Listing 25.6, inspired by examples from Conor Hoekstra, illustrates what the views do.¹

```
// https://godbolt.org/z/hYTv6f9eo
#include <ranges>
#include <array>
#include <string>
```

¹ *New Algorithms in C++23*, Conor Hoekstra, <https://youtu.be/VZPKHqeUQqQ?si=ucFmA2chl2IAgt2U&t=524>, CppNorth 2023, 2023-10-10, [2023-10-22]

```
#include <iostream>
#include <string_view>
using namespace std::literals; using namespace std;

// Function for outputting all sorts of things
template <typename OBJ>
void print(OBJ&& obj, int level = 0) {
    if constexpr(std::ranges::input_range<OBJ>) { // range
        cout << '[';
        for (const auto& elem : obj) print(elem, level+1);
        cout << ']';
    } else if constexpr(requires (OBJ tpl){ std::get<0>(tpl);}) { // tuple/pair
        cout << "(";
        print(get<0>(obj), level+1); print(get<1>(obj), level+1);
        cout << ")";
    } else cout << obj; // element
    if (level == 0) cout << '\n';
}

int main() {
    using namespace std::views; // exceptionally for brevity
    auto const nums = array{0, 0, 1, 1, 2, 2};
    auto const animals = array{"cat"s, "dog"s};
    print(iota(0, 5) | chunk(2)); // Output: [[01][23][4]]
    print(nums | chunk_by(equal_to{})); // Output: [[00][11][22]]
    print(iota(0, 5) | slide(3)); // Output: [[012][123][234]]
    print(iota(0, 10) | stride(3)); // Output: [0369]
    print(repeat(8) | take(5)); // Output: [88888]
    print(zip_transform(plus{}, nums, nums)); // Output: [002244]
    print(zip(iota(0, 3), iota(1, 4))); // Output: [(01)(12)(23)]
    print(iota(0, 4) | adjacent<2>); // Output: [(01)(12)(23)]
    print(iota(0, 4) | pairwise); // Output: [(01)(12)(23)]
    print(iota(0, 4) | adjacent_transform<2>(plus{})); // Output: [135]
    print(iota(0, 4) | pairwise_transform(plus{})); // Output: [135]
    print(animals | join_with( '+' )); // Output: [cat+dog]
    print(cartesian_product(iota(0, 2), "AZ"s)); // Output: [(0A)(0Z)(1A)(1Z)]
    print(enumerate("APL"s)); // Output: [(0A)(1P)(2L)]
    return 0;
}
```

Listing 25.6 Especially the C++23 views can be well combined.

The print function accepts both individual values as well as ranges and tuples. With the `fmt` library, this would be easier, but it is not available to me yet. Note the use of `if`

`constexpr` for a compile-time check. requires checks if `get<>` is available. This is the case for tuples and pairs, but also for some ranges, which is why the range check must come first.

25.6.2 Ranges as Parameters (and More)

However, I want to point out one thing. When you outsource code into a function that takes a range as a parameter, you typically write it in a compact form like this:

```
void print(auto&& range) {  
    for (const auto& elem : range) cout << elem; cout << '\n';  
}
```

Or, if you want to constrain the type to a suitable range concept, like this:

```
void print(ranges::input_range auto&& arg)  
    for (const auto& elem : range) cout << elem; cout << '\n';  
}
```

The type is specified with `auto&&`. The `auto` stands for an *abbreviated function template*. This is definitely a win, because otherwise, we would have to write a complicated template header for every function that takes a range as a parameter. This way is much better, and we don't even notice that we are defining a template.

But why don't I write `const auto&`? Why do I leave out `const`, and why do I use the (somewhat obscure) universal reference `&&`?

I had written something about caching views in the explanation for [Table 25.2](#), especially for `begin()`. However, caching means that using `begin()` modifies the view! Unfortunately, this is necessary to keep the cost of calling `begin()` low. The standard library promises that the amortized cost of `begin()` is $O(1)$ —that is, the cost for multiple calls. Even if the first call costs $O(n)$, caching can ensure that the average cost remains $O(1)$. Therefore, we unfortunately have to do without `const`.

And why `auto&&`? Isn't `auto&` sufficient? That would only work if we had a concrete variable to pass (an lvalue). But if we chain things using `operator|` in the call—as in several places in the example—then we have created a temp-value. So, to be able to pass both variables and temp-values, we need a universal reference—`auto&&`.

Because views do not work consistently, you might fall into the following trap. Take a look at the compiler error message for the following code.

```
// https://godbolt.org/z/3EbsPczh1  
void print(const auto& range) { // ✗ critical: constant reference  
    for (auto const& e : range) { cout << e; } cout << '\n';  
}
```

```
vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
list lst{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
print(vec);           // works on direct containers
print(lst);           // works on direct containers
print(vec | vs::take(3)); // take with vector works
print(lst | vs::take(3)); // take with list works
print(vec | vs::drop(5)); // drop with vector works
print(lst | vs::drop(5)); // ✗ drop with list does not work!
```

Listing 25.7 Functions must receive a range parameter as a universal reference.

The error on g++ 13.2 is this:

as 'this' argument discards qualifiers

This means that we made a `const` error. `drop` wants to cache the call to `begin()` and therefore modifies the view. That's why we need `auto&&` as a parameter.

Alternatively, the range parameter can also be `auto` without a reference—that is, a value parameter. But then the function would suddenly become very expensive if you called it for a real container. To prevent this, restrict the parameter with the `view` concept.

```
// https://godbolt.org/z/s1GjfWxv6
void print(ranges::view auto range) { // Value parameter, restricted to Views
    for (auto const& e : range) { cout << e; } cout << '\n';
}
vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
list lst{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
print(vec);           // ✗ Forbidden for containers, very good!
print(vs::all(vec));  // Convert container to view
print(vs::all(lst));  // Convert container to view
print(vec | vs::take(3)); // take with vector works
print(lst | vs::take(3)); // take with list works
print(vec | vs::drop(5)); // drop with vector works
print(lst | vs::drop(5)); // as a value parameter, drop with list works
```

Listing 25.8 A function only for views and not for containers.

So if you pass the view as a value parameter and for safety reasons forbid real containers as parameters, there are no surprises here.

I have shown you how to deal with the `const` trap that caching the views creates. Views can also be a bit tricky in other places. If you do not trust the standard views or have had bad experiences with them, you can try *Bellevue*s by Nico Josuttis (<https://github.com/josuttis/bellevue>), which are very similar to the standard views but do not cache for `begin()` and therefore do not have these problems.

25.6.3 List of Nonmodifying Algorithms

In this section, you will find functions that mainly query something from a range. With `find_if`, for example, you can search for the first occurrence of an element with a certain property in a range. You test this with a *predicate*, a function that returns `bool`. In the case of `find_if`, this is a unary predicate, meaning it is a function with one argument.

```
// https://godbolt.org/z/W9n7sTovz
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
#include <string_view>
#include <ranges>
using std::vector; using std::string; using std::string_view;
using namespace std::literals; using std::find_if;
vector<string> demo_split(string_view s) {
    vector<string> result{};
    auto it = s.begin();
    while(it != s.end()) {
        // until normal character:
        it = find_if(it, s.end(), [](char c) { return c!=' '; });
        // until whitespace:
        auto jt = find_if(it, s.end(), [](char c) { return c==' '; });
        if(it!=s.end())
            result.push_back(string(it, jt)); // Copy to result
        it = jt;
    }
    return result;
}
int main() {
    auto text = "The text is short"sv;
    auto res = demo_split(text);
    std::ranges::for_each(res, [](const string &e) {
        std::cout << "[" << e << "]" "; });
    std::cout << '\n'; // Output: [The] [text] [is] [short]
    // or directly with views::split:
    for(auto word : text | std::views::split(" "sv)) {
        std::cout << "[";
        for(auto c : word) std::cout << c;
        std::cout << "]" ";
    } // Output: [The] [text] [is] [short]
}
```

Listing 25.9 Searching with a predicate.

Here I use `find_if` first to skip spaces before a word in a string. `find_if` takes—like all algorithms—a range in which it should search. The range is defined by two iterators, the beginning and the end. The end always points to an element *behind* the last element of the range, so the empty range can always be specified by two identical iterators. Also useful: the number of elements in the range is simply the distance between the two iterators.

The last parameter in `find_if` is the *if*, which is the predicate. Here, I define a lambda that checks if the `char` character of the string is not a whitespace, meaning the start of the next word is found. The second call to `find_if` uses a comparison with a whitespace as the predicate, which is after the end of the word. I then copy the range enclosed by these two positions as a new string into the `result` using `push_back`.

However, with the introduction of ranges in C++20, a new `split` function was added, which makes the process somewhat simpler. Instead of returning a vector of string, the return type of `split` is `word`, which is a `ranges::subrange` within `text`. I then output it character by character using `c`.

In another example, I would like to demonstrate the good interaction of iterators with algorithms. Cleverly combined, you get powerful tools. The function `isPalindrome` checks if a string reads the same forward and backward. You can implement this as a one-liner.

```
// https://godbolt.org/z/MKxnGdrbr
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
#include <string_view>
using std::string_view; using namespace std::literals; using std::cout;
auto isPalindrome(string_view sv) {
    return std::ranges::equal(sv.begin(), sv.end(), sv.rbegin(), sv.rend()); };
int main() {
    for(auto s : {"madam"sv, "aibohphobia"sv, "abrakadabra"sv}) {
        cout << s << " is " << (isPalindrome(s)?"a":"not a") << " palindrome\n";
    }
}
```

Listing 25.10 Recognizing a palindrome with a single line of code.

The `equal` algorithm takes a range, through which it iterates forward, as its first two parameters. The third parameter is the beginning of a second range, with which the first range is then compared element-wise using `==`. If all elements are equal, `equal` returns true; at the first difference, it returns false. The clever part here is that the

beginning of the second range is passed with `s.rbegin()`. `rbegin()` returns an iterator that, when incremented with `++`, moves one element backward. So the same sequence is traversed backward here. In the end, the first letter is compared with the last, then the second with the second-to-last, and so on.

We don't necessarily have to restrict `isPalindrome` to `string_view`. We could also allow `input_range auto&&` as a parameter, and the function would work with any range and container that supports `rbegin()`. If I were more careful, I wouldn't traverse the entire sequence from start to finish, but only up to the halfway point, because `==` should be symmetric. I could achieve this by writing the following instead of `s.end()`: `s.begin()+(s.end()-s.begin())/2`.

In the case of `for_each`, information is not the goal; rather, a unary function without a return value is called on each element. Actually, these functions are intended to be read-only. However, if you provide an (in)appropriate functor or predicate, you can indeed modify the elements of the range. Doing so is not forbidden but probably confusing. With `for_each`, you see an example in [Listing 25.11](#). Instead, you should use [transform](#) [here](#).

```
// https://godbolt.org/z/jz1Gqo7d9
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> c{ 1,2,3,4 };
    std::ranges::for_each(c, [](auto &n) { n*=n; }); // ✎ modifying
    std::cout << c[3] << '\n'; // Output: 16
}
```

Listing 25.11 Even “read-only” algorithms like “`for_each`” can modify elements.

That is why I want to emphasize one thing: do not use `for_each` to solve every problem. There are often other algorithms that are better at solving certain problems. And if you look at the list of algorithms from time to time, you will become more comfortable with them over time, your code will become more compact and—I claim—better. A few examples:

- If you want to determine how often an element is contained in a sequence, use `count`, not `for_each`.
- If you want to check a predicate for all elements, use `all_of`.
- If you want to know if at least one element satisfies a predicate, use `any_of`.
- If a sequence contains the same elements as another, just in a different order, by no means use `for_each`; instead, use `is_permutation`.

These are the available nonmodifying algorithms:

- **all_of**
Usage: frequent; iterators: in
Checks if a predicate is true for all elements.
- **any_of**
Usage: frequent; iterators: in
Checks if at least one element satisfies the predicate.
- **none_of**
Usage: frequent; iterators: in
Checks if none of the elements satisfy a predicate.
- **for_each**
Usage: constant; iterators: in
Applies a unary function to each element, see [Listing 25.11](#).
- **for_each_n**
Usage: rare; iterators: in
Applies a unary function to the first *n* elements of a range.
- **find**
Usage: constant; iterators: in
Finds by a sample value.
- **find_if**
Usage: constant; iterators: in
Finds by a predicate, see [Listing 25.9](#).
- **find_if_not**
Usage: frequent; iterators: in
Finds by the negation of a predicate.
- **find_end**
Usage: rare; iterators: fw
Finds the end of a matching subsequence.
- **find_first_of**
Usage: frequent; iterators: in/fw
Finds one of several sample values.
- ***ranges::find_last, find_last_if, find_last_if_not***
Usage: occasional (C++23); iterators: in/fw
Finds the last element or the element matching the predicate.
- **adjacent_find**
Usage: occasional; iterators: fw
Finds the first position where two adjacent elements both satisfy a predicate.

- **count**
Usage: constant; iterators: in
Counts the number of occurrences of a sample value.
- **count_if**
Usage: constant; iterators: in
Counts how often a predicate matches.
- **mismatch**
Usage: frequent; iterators: in
Finds the first difference between two sequences.
- **equal**
Usage: constant; iterators: in
Checks if all elements are pairwise equal, see [Listing 25.10](#).
- **is_permutation**
Usage: occasional; iterators: in
Is one sequence just a permutation of another?
- **search**
Usage: frequent; iterators: fw
Finds the position where a subsequence is hidden within a larger sequence.
- **search_n**
Usage: occasional; iterators: fw
Finds the position with *n* consecutive values equal to the sought value.
- ***range::contains*, *ranges::contains_subrange***
Usage: frequent (C++23); iterators: fw
Checks whether a single element or an entire range is contained within another range. *contains* is like `find(...) != end()`, *contains_subrange* is like `search(...)`, but more expressive.
- ***ranges::starts_with***
Usage: frequent (C++23); iterators: fw
Checks if one range starts with another.
- ***ranges::ends_with***
Usage: frequent (C++23); iterators: fw
Checks if one range ends with another.

`std::search` optionally takes an additional `Searcher` parameter, which specifies the algorithm to be used for searching a subsequence in a container (e.g., `string`):

```
template<forward_iterator It, class Searcher>  
ForwardIterator std::search(It first, It last, const Searcher& searcher);
```

At least the well-known Boyer-Moore algorithms like `boyer_moore_searcher` and Boyer-Moore-Horspool algorithms like `boyer_moore_horspool_searcher` are available. These

can significantly speed up the search, especially when the search text is long and consists of many different characters. Do not expect a significant speedup when searching for "aaaaaa" or "abababab". You can also let the implementation decide by using `default_searcher`. The `std::ranges::search` variant does not support the `Searcher` parameter.

25.6.4 Inherently Modifying Algorithms

The functions in this section write a result to an output range. For some of the algorithms, it is allowed for the output range and input range to overlap. This is not always the case, so be careful and check the documentation to see if overlapping is allowed. Some algorithms like `unique` modify the sequence in place.

Let me highlight a small group of functions that are related but do not appear directly next to each other due to alphabetical sorting—or even in a different list. The following functions all work in some form on input and/or output sequences and apply a function `f` defined by you to the elements:

- `for_each` applies `f` to each element of an input sequence; `f` returns `void` and is unary.
- `generate` applies `f` to each element of the output sequence; `f` returns an element and is nullary.
- `transform` applies `f` to each element of the input sequence and writes the result to an output sequence; `f` returns an element and is unary.
- `transform` can also apply `f` to two input sequences and write the result to an output sequence; in this case, `f` returns an element and is binary.

By its nature, `for_each` usually triggers a side effect, such as output or changing an internal state. Similarly, `generate` will likely change an internal state as a side effect; otherwise the generated sequences would be quite boring.

Especially `transform` is extremely powerful. `f` can indeed accept arguments of a completely different type than it returns. The elements of the containers underlying the iterators only need to match `f`. In [Listing 25.12](#), I transform a sequence of `char` and `int` to a sequence of `string`.

```
// https://godbolt.org/z/7r4b8jdGr
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
using std::to_string; using std::string; using std::vector;
struct Squares {
    mutable int n = 1;
    int operator()() const { return n*n++; }
};
```

```
int main() {
    vector<int> sq(10);
    std::ranges::generate(sq, Squares{});
    std::ranges::for_each(sq, [](auto n) {
        std::cout << n << " "; });
    std::cout << '\n';           // Output: 1 4 9 16 25 36 49 64 81 100
    string a = "NCC-";
    vector<int> b {1,7,0,1};
    vector<string> c(4);
    auto f = [](char c, int i) -> string { return c+to_string(i); };
    std::ranges::transform(
        a,           // Input 1
        b,           // Input 2
        c.begin(),   // Output
        f);          // string f(char,int)
    std::ranges::for_each(c, [](auto s) {
        std::cout << s << " "; });
    std::cout << '\n';           // Output: N1 C7 C0 -1
}
```

Listing 25.12 “transform” can juggle with different types.

The example around `generate` demonstrates how functor `Squares` continuously generates square numbers. `sq` contains 10 elements, so the functor is called 10 times. Because it maintains an internal state with `n`, but the `operator()` must be `const`, the `n` is marked as `mutable` to be changeable.

In the `transform` algorithm, `a` holds elements of type `char`, `b` holds elements of type `int`, and the output goes into `c` with elements of type `string`. Therefore, for `transform`, the function `f` must have the `function<string(char,int)>` type, which is the case for the lambda `f`.

For the `transform` example, you can also use range adapters.

```
// https://godbolt.org/z/j9ozax8qE
#include <ranges> // zip_transform
// ...
auto res = std::views::zip_transform(f, a, b); // Range adapter
for(auto s: res) { std::cout << s << " "; };
std::cout << '\n';           // Output: N1 C7 C0 -1
```

Listing 25.13 Transforming with range adapters.

Unlike the `std::transform` and `std::ranges::transform` algorithms, the range adapter `std::views::zip_transform` works lazily. It can also handle more than two sequences, which is why the operation `f` is specified first. `zip_transform` was introduced in C++23:

- **copy**
Usage: constant, iterators: in/out
Copies a range element-wise into a target range.
- **copy_n**
Usage: frequent, iterators: in/out
Copies *n* elements from a position.
- **copy_if**
Usage: frequent, iterators: in/out
Copies elements only if they satisfy a predicate.
- **copy_backward**
Usage: occasional, iterators: bi
Effectively like *copy*, but starts from the end; important for overlapping copies.
- **move**
Usage: frequent, iterators: in/out
Like *copy*, but moves elements.
- **move_backward**
Usage: occasional, iterators: bi
Like *copy_backward*, but moves elements.
- **swap**
Usage: constant, iterators: –
Swaps two *values* (not ranges).
- **swap_ranges**
Usage: frequent, iterators: fw
Swaps the values of two ranges element-wise.
- **iter_swap**
Usage: occasional, iterators: fw
Swaps the values pointed to by two iterators (not ranges).
- **transform**
Usage: frequent, iterators: in/out
Like *copy*, but applies a unary function to each element. *transform* can also take two input sequences and a binary function. The output range can be identical to one of the input ranges, as shown in [Listing 25.12](#).
- **replace**
Usage: frequent; iterators: fw
Writes a new value to all positions in a sequence that match a given value.
- **replace_if**
Usage: frequent; iterators: fw
Writes a value to all positions in a sequence that satisfy a predicate.

- **replace_copy**
Usage: occasional; iterators: in/out
Like a mix of `copy` and `replace` at the destination.
- **replace_copy_if**
Usage: occasional; iterators: in/out
Like a mix of `copy` and `replace_if` at the destination.
- **fill**
Usage: constant; iterators: fw
Fills a range with a value.
- **fill_n**
Usage: constant; iterators: out
Fills `n` values starting from a point.
- **generate**
Usage: constant; iterators: fw
Executes a zero-argument functor element-wise in a target range.
- **generate_n**
Usage: constant; iterators: out
Executes a zero-argument functor element-wise `n` times starting from a position.
- **remove**
Usage: constant; iterators: fw
Moves all occurrences of a value to the end of the sequence.
- **remove_if**
Usage: constant; iterators: fw
Moves all elements for which a predicate is true to the end of the sequence.
- **remove_copy**
Usage: occasional; iterators: in/out
Copies elements to a target position unless they are equal to a specified value.
- **remove_copy_if**
Usage: frequent; iterators: in/out
Copies elements to a target position unless they satisfy a specified predicate. I find the name “remove” particularly misleading here, as it is more of a “filtered copy”, and I find that extremely useful.
- **unique**
Usage: constant; iterators: fw
Adjacent equal elements are pushed to the end; see [Listing 25.3](#).
- **unique_copy**
Usage: constant; iterators: in/out
Always copies only the first of several adjacent equal elements.

- **reverse**
Usage: constant; iterators: in/out
Reverses a range in place.
- **reverse_copy**
Usage: frequent; iterators: bi/out
Creates a reversed copy of a range.
- **rotate**
Usage: occasional; iterators: fw
Rotates the elements within a range so that a selected element is placed at the beginning.
- ***ranges::shift_left, ranges::shift_right***
Usage: occasional (C++23); iterators: fw
Shifts elements within a range by an offset.
- **rotate_copy**
Usage: occasional; iterators: fw/out
Like rotate, but the target can be a different range.
- **sample**
Usage: frequent; iterators: ra
Selects a random subset from a larger set.
- **shuffle**
Usage: frequent; iterators: ra
Randomly arranges the elements of a sequence.

With the `sample` function, you can more easily perform the recurring task of selecting a random sample from a set:

```
// https://godbolt.org/z/s53qx664K
#include <iostream>
#include <random>           // default_random_engine
#include <string>
#include <iterator>         // back_inserter
#include <algorithm>        // sample

int main() {
    std::default_random_engine random{};
    const std::string in = "abcdefgh";
    for(auto idx : {0,1,2,3}) {
        std::string out;
        std::ranges::sample(in, std::back_inserter(out), 5, random);
        std::cout << out << '\n';
    }
}
```

The `in` variable represents the population, and the samples are output to the variable `out`. The number of samples is set to 5, indicating that five letters are always output. The loop from 0 to 3 performs this experiment multiple times.

The output (on my system, by chance) is this:

```
abcfg  
abcde  
bdfgh  
abefg
```

As you can see, `sample` ensures that the elements of the sample retain their order.

25.6.5 Algorithms for Partitions

A *partition* is the division of a range into two subranges based on a predicate. Everything in the left subrange matches the predicate; everything in the right range does not match the predicate. The point between the two is the *partition point*.

The standard library provides a modest selection of algorithms related to partitions. These algorithms are useful in a variety of contexts and can be used to implement highly efficient sorting algorithms in a general manner. The algorithms require that the iterators originate from a container with bidirectional iterators, not necessarily random-access iterators. In fact, a simple form of *quicksort* can be implemented with this type of partition:

- **is_partitioned**
Usage: occasional; iterators: in
Checks if a range is already neatly partitioned based on a predicate.
- **partition**
Usage: frequent; iterators: bi
Partitions a range into two subranges based on a predicate.
- **stable_partition**
Usage: occasional; iterators: bi
Like `partition`, but maintains the relative order of elements within each subrange.
- **partition_copy**
Usage: frequent; iterators: in/out
Copies elements that satisfy a predicate to one destination and the others to another destination.
- **partition_point**
Usage: rare; iterators: fw
Finds the first element in a range for which a predicate does not hold, but only if it is a valid partition.

25.6.6 Algorithms for Sorting and Fast Searching in Sorted Ranges

Searching and sorting are extremely important. Doing this skillfully is a good way to achieve fast programs. Sometimes, it is best to simply use an automatically sorted container like `multimap` or `set` or the *flat_set* adapters introduced in C++23. Very often, however, this is not the best method. First, put all elements, as they come, into any sequence container and then sort them all at once. `map` is not very memory-friendly, and the elements are scattered all over. A `vector` behaves much better in this regard and works excellently with the algorithms for sorting and *binary searching*. The *flat_...* adapters around a `vector` are a compromise, but the best approach is to first collect the data, then sort:

- **sort**
Usage: constant; iterators: ra
Sorts the elements in the range; see for example [Listing 25.2](#).
- **stable_sort**
Usage: frequent; iterators: ra
Like `sort`, but ensures that equal elements retain their relative order.
- **partial_sort**
Usage: occasional; iterators: ra
Like `partition`, uses the comparison with the middle element as the predicate.
- **partial_sort_copy**
Usage: occasional; iterators: in/ra
Like `partial_sort`, but copies the result into a target range.
- **is_sorted**
Usage: frequent; iterators: fw
Checks if a range is sorted.
- **is_sorted_until**
Usage: occasional; iterators: fw
Finds the first element in a range that breaks the sorting.
- **nth_element**
Usage: occasional; iterators: ra
Sorts the elements in the range, but only until the *n*-th element is determined.
- **lower_bound**
Usage: constant; iterators: fw
Identifies the first element within a sorted range.
- **upper_bound**
Usage: frequent; iterators: fw
Finds the last sought element in a sorted range.
- **equal_range**
Usage: frequent; iterators: fw
Corresponds to `lower_bound` and `upper_bound` simultaneously.

■ `binary_search`

Usage: occasional; iterators: fw

Like `lower_bound`, but returns *true* if the element is found.

25.6.7 Set Algorithms Represented by a Sorted Range

With *merge* algorithms, you can also solve some tricky tasks. Clever sorting using *Mergesort* is an example of this, one suitable for extremely large datasets, among other things, because these algorithms work with input and output iterators, the least demanding groups.

The family of merge algorithms can do much more. For example, you can simulate the functionality of *mathematical sets* with sorted ranges—and that without having to use a C++ set, because a vector or even a list is sufficient. However, caution: If the sorted range contains duplicates, this does not yet correspond to the mathematical set, which is why I call this range a *merge set*. But if you remove duplicates beforehand and then use the `set...` functions, you can simulate real sets.

```
// https://godbolt.org/z/1esod6hrT
#include <algorithm>
#include <iostream>
#include <list>
#include <string>
#include <iterator> // ostream_iterator
#include <cctype>    // toupper
using std::toupper;
int main() {
    std::list a{ 1,2,4,4,4,7,9 };
    std::list b{ 2,2,3,4,4,8 };
    using Os = std::ostream_iterator<int>; // Type of output iterator
    Os os{std::cout, " "};                // Stream output iterator for int
    auto run = [&a,&b,&os](auto algo) {    // use a, b, and os
        algo(a.begin(), a.end(), b.begin(), b.end(), os); // Call algorithm
        std::cout << '\n';
    };

    // Results of the algorithms
    using It = decltype(a.begin());       // Type of input iterators

    run(std::merge<It,It,Os>);             // Output: 1 2 2 3 4 4 4 4 7 7 8 9
    run(std::set_union<It,It,Os>);         // Output: 1 2 2 3 4 4 4 7 7 8 9
    run(std::set_intersection<It,It,Os>); // Output: 2 4 4
    run(std::set_difference<It,It,Os>);   // Output: 1 4 7 7 9
    run(std::set_symmetric_difference<It,It,Os>); // Output: 1 2 3 4 7 7 8 9
```

```
// With letters it becomes even clearer
std::string x = "abdddggi";
std::string y = "BBCDDH";
using Us = std::ostream_iterator<char>; // Type of output iterator
Us us{std::cout, ""}; // Stream output iterator for char
auto compare = [](auto c, auto d) { return toupper(c) < toupper(d); };
auto run2 = [&x,&y,&us,&compare](auto algo) { // use x,y, and us
    algo(x.begin(), x.end(), y.begin(), y.end(), us, compare);
    std::cout << '\n';
};
// Results of the algorithms
using Jt = decltype(x.begin()); // Type of input iterators
using Cm = decltype(compare); // Type of comparison function

run2(std::merge<Jt,Jt,Us,Cm>); // Output: abBBCdddDDggHi
run2(std::set_union<Jt,Jt,Us,Cm>); // Output: abBCdddggHi
run2(std::set_intersection<Jt,Jt,Us,Cm>); // Output: bdd
run2(std::set_difference<Jt,Jt,Us,Cm>); // Output: adggi
run2(std::set_symmetric_difference<Jt,Jt,Us,Cm>); // Output: aBCdggHi
}
```

Listing 25.14 Functionality of the set algorithms.

I sequentially apply the set algorithms that have two input sets and one output set. This happens in the `run` and `run2` lambdas. If you do not specify a comparison functor—shown in `run`—then it will compare based on `<`. For the second example, I defined a comparison operator `compare` that ignores case sensitivity. This way, you can see in the results which original set each output character comes from:

- **merge**
Usage: frequent; iterators: in/out
Merges two sorted ranges into a new sorted range.
- **inplace_merge**
Usage: occasional; iterators: bi
Like `merge`, but assumes the ranges are adjacent.
- **includes**
Usage: occasional; iterators: in
Checks if set B is a subset of set A.
- **set_union**
Usage: occasional; iterators: in/out
Merges two containers, A and B; duplicates in B are retained; skips elements in B that are present in A.

- **set_intersection**
Usage: occasional; iterators: in/out
Copies elements from A that are also present in B.
- **set_difference**
Usage: occasional; iterators: in/out
Copies elements from A that do not occur in B.
- **set_symmetric_difference**
Usage: occasional; iterators: in/out
Copies elements from A and from B that do not occur in the other set.

25.6.8 Heap Algorithms

A *heap* in computer science is a special data structure that is not inherently sorted but shares certain properties with a sorted range. In a heap, you can efficiently remove the largest element or insert any element at any time. Does this sound almost like sorting? It is. *Heap-sort* is one of the best sorting algorithms, and it uses exactly this data structure. The elements in the range that form the heap do not reveal much at first glance. However, with `sort_heap` you can quickly convert a heap into a sorted range.

All heap algorithms require random-access iterators. This limits the choice of containers. A `vector` is ideal:

- **make_heap**
Usage: rare; iterators: ra
Rearranges elements into a valid heap.
- **push_heap**
Usage: rare; iterators: ra
Adds an arbitrary element to the heap.
- **pop_heap**
Usage: rare; iterators: ra
Removes the largest element from the heap.
- **sort_heap**
Usage: rare; iterators: ra
Converts a heap into a sorted range.
- **is_heap**
Usage: rare; iterators: ra
Checks if a range is a valid heap.
- **is_heap_until**
Usage: rare; iterators: ra
Finds the first element that breaks the heap property.

25.6.9 Minimum and Maximum

Minimum and maximum often need to be calculated, sometimes from a fixed set of elements, sometimes from a container:

- **min**
Usage: constant; iterators: –
Returns the smallest element of two or from an initializer list.
- **max**
Usage: constant; iterators: –
Returns the largest element of two or from an initializer list.
- **minmax**
Usage: constant; iterators: –
Returns `min` and `max` as a pair.
- **clamp**
Usage: frequent; iterators: –
Returns a value that is guaranteed to lie between two bounds.
- **min_element**
Usage: constant; iterators: fw
Returns an iterator (!) to the smallest element of a range.
- **max_element**
Usage: constant; iterators: fw
Returns an iterator (!) to the largest element in a range.
- **minmax_element**
Usage: frequently; iterators: fw
Returns a pair of iterators for `min_element` and `max_element`.

25.6.10 Various Algorithms

With `lexicographical_compare`, you can compare ranges like phone book entries. `{1,1,2}` is smaller than `{1,2,1}`.

For combinatorics—or solving puzzle tasks—it is very useful to be able to try all permutations of an input. If the elements of the range support less than `<`, you can transform a range with n elements into its next permutation with a call to `next_permutation`. This algorithm internally uses `lexicographical_compare`. After $n!$ permutations, you will return to the original permutation:

```
// https://godbolt.org/z/j6GPKc36P
#include <iostream>
#include <algorithm>
#include <string>
```

```

void one(std::string &seq) {
    std::ranges::next_permutation(seq);
    std::cout << seq << '\n';
}

int main() {
    std::string seq = "JQK";
    std::cout << seq << '\n'; // Output: JQK
    auto limit = 3*2*1;        // n!
    for(int i=0; i<limit; ++i)
        one(seq);
    // Here the sequence is back to its original state.
}

```

The output of this program is as follows:

```

JQK
KJQ
KQJ
QJK
QKJ
JKQ
JQK

```

So exactly the six different permutations of "JQK", and the seventh is again the original sequence.

These are the various algorithms you can use:

- **lexicographical_compare**
Usage: frequent; iterators: in
Compares two ranges element-wise with less.
- **next_permutation**
Usage: occasional; iterators: bi
The next of $n!$ permutations of the range.
- **prev_permutation**
Usage: occasional; iterators: bi
The previous of $n!$ permutations of the range.

25.7 Element-Linking Algorithms from “<numeric>” and “<ranges>”

The <numeric> header contains some functions that are useful in the context of numerical calculations. The functions work together with pairs of iterators or ranges. They are used on containers (or streams) that contain numerical values—that is, support arithmetic operators.

Some algorithms have received more general variants of these functions with C++23 and ranges in the `<ranges>` header in the `std::ranges::` namespace and are summarized as *fold operations*. They all start with *fold_* in their names. In the following list, you will see a reference to the more specific function in the corresponding description of the *fold* function. For example, *fold_left* is a more general form of *accumulate*. The remaining classical algorithms can be mapped using range adapters and are thus found in the `std::views::` namespace.

In the overview, I write `std::` before the function names when it is a classical algorithm from the header `<numeric>`, and `rs::` for the new functions from `<ranges>`. In the text, for the sake of clarity, I prefix the range adapters with `vs::`.

■ **`std::accumulate`**

Usage: constant; iterators: in

This function is suitable for calculating `r = init # a # b # c # ...`. Here, `#` is any binary function or operator+ if you do not specify one. `init` represents the “neutral” element with respect to the operation (for example, 0 for addition or 1 for multiplication). Subexpressions are always evaluated from left to right, so you must use `reduce` for parallel execution. You can see an example in [Listing 25.15](#).

■ **`std::reduce`**

Usage: constant; iterators: in

Like `accumulate`, except the grouping is not specified. Therefore, use only associative operators like addition and multiplication. Subtraction, for example, is not associative. Due to this property, `reduce` is particularly suitable for execution: `:par`; see [Listing 25.4](#).

Do not underestimate this function. You might have heard of it in another context, perhaps from “MapReduce” or *fold*, *aggregate*, or *compress* from functional languages.

■ **`rs::fold_left`, `rs::fold_left_first`**

Usage: frequent (C++23); iterators: in

Like `accumulate`, but you always specify the binary operation. The `..._first` variants take the first value of the range as the initial value. For the variants without `..._first`, you additionally provide the initial value. You get the result of the computation back.

■ **`rs::fold_left_with_iter`, `rs::fold_left_first_with_iter`**

Usage: occasional (C++23), iterators: in

Like `fold_left` and `fold_left_first`, but returns an iterator.

■ **`std::adjacent_difference`**

Usage: occasional; iterators: in

This outputs the results of the operations `a#b`, `b#c`, `c#...` etc. into an output container. If you do not provide a binary operator, `operator-` is used. You get a kind of ‘discrete derivative’ of the values in the container. The output container must have space for `n-1` elements; see [Listing 25.16](#). For ranges, since C++23, use `vs::adjacent_transform`.

- **std::inner_product**

Usage: occasional; iterators: in

The *inner product* of two ranges $\{a, b, c, \dots\}$ and $\{x, y, z, \dots\}$ is $r = a*x + b*y + c*z + \dots$. As always, you can also choose the default operators $+$ and $*$ yourself. The result is a single 'accumulated' value. The order of operations is from left to right; for parallel execution, use `transform_reduce`.

- **std::transform_reduce**

Usage: rare; iterators: in

First applies a functor to each element before `reduce` is executed. Some overloads take two input ranges, allowing you to link multiple containers, resulting in a parallel variant of `inner_product`, as the order is not specified. Not equivalent, but the C++23 `vs::adjacent_transform_range` adapter and its `vs::pairwise_transform` specialization work similarly.

- **std::partial_sum**

Usage: frequent; iterators: in/out

Similar to `accumulate`, elements are summed sequentially (or combined with an operator you specify), but here the intermediate results are also written to an output iterator. The first element of the output is then a , the second $a+b$, the third $a+b+c$, etc. The evaluation order is always from left to right, parallel variants are the `..._scan` functions.

- **std::exclusive_scan**

Usage: rare; iterators: in/out

Like `partial_sum`, where the i -th intermediate result is without the i -th element. The order of operations is not fixed, but it is parallelizable.

- **std::inclusive_scan**

Usage: rare; iterators: in/out

Like `partial_sum`, where the i -th intermediate result includes the i -th element. The difference from `exclusive_scan` is shown in [Listing 25.18](#).

- **std::transform_exclusive_scan**

Usage: rare; iterators: in/out

Like `exclusive_scan`, but applies a function to each element beforehand.

- **std::transform_inclusive_scan**

Usage: rare; iterators: in/out

Like `inclusive_scan`, but applies a function to each element beforehand.

- **std::iota, vs::iota**

Usage: constant; iterators: fw

This allows you to generate a sequence of values in an output range. You start with `init`, to which $++$ is successively applied until the end of the range is reached; see [Listing 25.17](#).

Exception: Algorithms on Numbers

Three functions in `<numeric>` do not match the pattern as they work on exactly two values and not on iterators:

- `gcd(n, m)`
Returns the *greatest common divisor (gcd)* of two numbers.
- `lcm(n, m)`
Returns the *least common multiple (lcm)* of two numbers.
- `midpoint(n, m)`
Determines the midpoint (rounded to *n*) between *n* and *m* starting from C++20. No overflow occurs, which could happen with $(n+m)/2$, for example. Also works on pointers. For linear interpolation with floating-point numbers, use `lerp` from `<cmath>`.

All these functions are `constexpr` and thus fast with numeric literals. Therefore, they are well-suited for initializing constants.

Consider the following examples.

```
// https://godbolt.org/z/Y7GPcqWM8
#include <numeric>    // accumulate
#include <functional> // multiplies
#include <algorithm>  // transform, fold_left
#include <iostream>
#include <vector>
using std::accumulate; using std::cout; using std::vector;
using std::multiplies;
namespace rs = std::ranges;
int main() {
    vector data{ 2, 3, 5, 10, 20 };
    cout << accumulate(data.begin(), data.end(), 0) << '\n';    // +, Output: 40
    cout << rs::fold_left(data, 1, multiplies<int>{}) << '\n'; // *, 6000
    vector<bool> even( data.size() );
    std::transform( data.begin(), data.end(), even.begin(),
        [](auto n) { return n%2==0; });
    for(auto b : even) {
        cout << ( b ? "even " : "odd ");
    }
    cout << "\n";    // Output: even odd odd even even
    auto areAllEven = accumulate(even.begin(), even.end(), true,
        [](auto b, auto c) { return b&& c; });
    if(areAllEven) {
        cout << "all even numbers\n";
    }
}
```

```

    } else {
        cout << "odd numbers included\n"; // this is the output
    }
}

```

Listing 25.15 “accumulate” example.

fold_left is available from C++23. Instead of multiplies you could also write `[](auto a, auto b){ return a*b; }`, and instead of the lambda for `allAreEven` you could also use `logical_and<bool>{}` from `<functional>`.

```

// https://godbolt.org/z/5Whz59374
#include <numeric>    // adjacent_difference
#include <functional> // plus
#include <algorithm>  // copy
#include <iostream>
#include <iterator>   // ostream_iterator
#include <vector>
#include <ranges>     // pairwise_transform
using std::cout; using std::vector; namespace vs = std::views;
int main() {
    // Stream output iterator for int;
    std::ostream_iterator<int> os{std::cout, " "};
    vector data{ 1, -1, 2, -2, -4, 4, -6, 6 };
    std::copy(data.begin(), data.end(), os);
    cout << '\n'; // Output: 1 -2 -2 -4 4 -6 6
    vector<int> res( data.size()-1 ); // Space for result
    // Write results to res:
    adjacent_difference(data.begin(), data.end(), res.begin());
    std::copy(res.begin(), res.end(), os);
    cout << '\n'; // Output: 1 -2 3 -4 -2 8 -10
    // Write directly to os:
    adjacent_difference(data.begin(), data.end(), os, std::plus<int>{});
    cout << '\n'; // Output: 1 0 1 0 -6 0 -2 0
    // or via range adapter:
    for(auto e: vs::pairwise_transform(data, std::plus<int>{}))
        cout << e << ' ';
    cout << '\n'; // Output: 0 1 0 -6 0 -2 0
}

```

Listing 25.16 “adjacent_difference” and “pairwise_transform”.

Instead of my usual for-loop for outputting a container, I have used a `ostream_iterator` *stream output iterator* here: you create an iterator not with `begin()` as an indirection

into a container, but into an output stream—here `cout` with the delimiter string " " between the elements. I use it to output data to the console.

Then I prepare the first `adjacent_difference` by creating space for the result. I first initialize the vector `res` with the appropriate size. Then I execute `adjacent_difference`—the pairwise differences of the elements from `data` end up in `res`.

But especially with functions that write outputs to iterators, preparing the output container is actually cumbersome. And when I actually just want to output the result anyway, I can also use the stream output iterator, as I do in the second example.

```
// https://godbolt.org/z/WPrP79sv6
#include <numeric>    // accumulate, iota
#include <algorithm>  // copy
#include <iostream>
#include <iterator>   // ostream_iterator
#include <vector>
#include <ranges>     // iota, take, stride
using std::accumulate; using std::cout; using std::vector;
namespace vs = std::views;
struct Generator {
    int state_;
    void operator++() { state_ += state_; }
    operator int() { return state_; }
};
int main() {
    std::ostream_iterator<int> os{std::cout, " "}; // Stream output iterator for int
    vector<int> data(7);
    std::iota(data.begin(), data.end(), 10);
    std::copy(data.begin(), data.end(), os);
    cout << '\n';           // Output: 10 11 12 13 14 15 16
    vector<int> seq(7);
    std::iota(seq.begin(), seq.end(), Generator{2});
    std::copy(seq.begin(), seq.end(), os);
    cout << '\n';           // Output: 2 4 8 16 32 64 128
    for(auto i: vs::iota(0) | vs::stride(3) | vs::take(6))
        cout << i << ' ';
    cout << '\n';           // Output: 0 3 6 9 12 15
}
```

Listing 25.17 “iota”, “stride”, and “take”.

Even though it looks like you can only generate sequences of ascending numbers here (which is already quite useful), with a suitable `operator++` you can generate quite interesting sequences. Helper type `Generator` stores a state that is retrieved during the

implicit conversion to `int`—specifically when `iota` wants to write the value to the output iterator. `operator++` on the helper type can execute any function; here I just double the value of the internal state. The result is an almost arbitrary output sequence that can be generated over a changing state.

At the end, you will see the `vs::iota` range generator. Here I start at 0 and generate infinitely many values. With `stride`, I only take every third value. Because the process is *lazy*, it can simply be stopped with `take` after the sixth value. The last two range adapters are new in C++23.

In the next listing, I perform `inclusive_scan` and `exclusive_scan` on a dataset.

```
// https://godbolt.org/z/enj3znvTs
#include <numeric> // *_scan
#include <iostream>
#include <vector>
using std::inclusive_scan; using std::exclusive_scan;
std::ostream& operator<<=(std::ostream&os, const std::vector<int>&data) {
    for(auto &e : data) os << e << ' '; return os << '\n';
}
int main() {
    std::vector data{ 1, 3, 10, 18, 30, 50 };
    std::vector<int> result(6);    // 6 elements
    auto plus = [](auto a, auto b) { return a+b; };
    inclusive_scan(data.begin(),data.end(), result.begin(), plus, 100);
    std::cout <<= result;
    // Output: 101 104 114 132 162 212
    exclusive_scan(data.begin(),data.end(), result.begin(), 100);
    std::cout <<= result;
    // Output: 100 101 104 114 132 162
}
```

Listing 25.18 “`inclusive_scan`” and “`exclusive_scan`”.

In `inclusive_scan`, the first element of the output already contains `100+1`, so the *i*-th result already includes the *i*-th input in the sum.

The result of `exclusive_scan` starts with 100, so the *i*-th result does not include the *i*-th input, thus it is *exclusive* of the *i*-th element. You always need an initial value for the first result here. I defined `plus` for `inclusive_scan` only so that I could use a comparable overload: if you want to specify an `init` value for `inclusive_scan`, you also need an operator as an argument.

25.8 Copy instead of Assignment: Values in Uninitialized Memory Areas

All functions from `<algorithm>` use assignment with `E& operator=(const E&)` when moving elements. With `move_iterator`, you can make an algorithm use the move assignment `E& operator=(E&&)` instead. But in both cases, an object must already exist at the destination. That means you need to initialize the entire target range with valid objects first. This can be cumbersome and unnecessary as you know that `copy` will overwrite the entire range anyway.

In the `<memory>` header, there are a handful of additional algorithms that *do not* require a valid object at the destination. The objects are then created with the copy constructor `E(const E&)` in *uninitialized memory*, not overwritten. The size of this memory must still be sufficient. The ranges variants are also available starting from C++20.

To obtain uninitialized memory, you can use `get_temporary_buffer` or the `malloc` or `alloca` C function. To create an object at a specific location, all these algorithms use *placement new*, a special form of `new` in which the programmer, not the compiler, decides where the object should be created.

Assume that the `int` element type is a data type that is really expensive to create, where it would be worthwhile to save the unnecessary initialization. Then you would use `uninitialized_copy` as follows:

```
// https://godbolt.org/z/dKPbj1f8
#include <iostream>
#include <memory>    // uninitialized_copy
#include <alloca.h>   // alloca (Linux)
#include <list>
int main () {
    const std::list input{1,9,2,6,6,6,8};
    const auto SZ = input.size();
    // uninitialized memory area:
    int* target = (int*)alloca(sizeof(int) * SZ); // space for 7 ints
    std::uninitialized_copy(input.begin(), input.end(), target);
    // test output
    for(int idx=0; idx<SZ; ++idx) {
        std::cout << target[idx] << ' ';
    }
    std::cout << '\n'; // output: 1 9 2 6 6 6 8
}
```

The call to `alloca` allocates space for seven `int` on the stack (which is automatically freed when the function exits). The memory area is uninitialized—default for the built-in `int` anyway, but unlike usual, real C++ objects would also be uninitialized.

Then the elements from `input` are copied to `target`—and this time with the copy constructor instead of the assignment operator. Therefore, it is okay that there are no valid objects at the target location. For real classes, this could make a difference.

By the way, `alloca` should only be used with extreme caution, so only in functions where nothing can go wrong.

- **`uninitialized_copy`**
Usage: special; iterators: in/fw
Copies elements of a range using the copy constructor.
- **`uninitialized_copy_n`**
Usage: special; iterators: in/fw
Copies *n* elements from a starting point to a target point using the copy constructor.
- **`uninitialized_fill`**
Usage: special; iterators: fw
Fills a range with a specific value using the copy constructor.
- **`uninitialized_fill_n`**
Usage: special; iterators: fw
Creates *n* copies of a value from a starting point using the copy constructor.
- **`uninitialized_move`**
Usage: special; iterators: fw
Moves elements into an uninitialized range.
- **`uninitialized_move_n`**
Usage: special; iterators: fw
Moves *n* elements into an uninitialized range.
- **`uninitialized_default_construct`**
Usage: special; iterators: fw
Fills a range with the values of the default constructor.
- **`uninitialized_default_construct_n`**
Usage: special; iterators: fw
Creates *n* objects using the default constructor.
- **`uninitialized_value_construct`**
Usage: special; iterators: fw
Fills a range with value-constructed objects.
- **`uninitialized_value_construct_n`**
Usage: special; iterators: fw
Creates *n* objects using value constructor.
- **`destroy_at`**
Usage: special; iterators: fw
Calls the destructor of an object at a specific address.

- **Destroy**
Usage: special; iterators: fw
Destroys a range of objects.
- **destroy_n**
Usage: special; iterators: fw
Destroys *n* objects.

25.9 Custom Algorithms

If you want to write such useful functions yourself that you store them in a header for the future of your programming career and then reuse them in all projects, it is quite simple to do. Well, you will need one or two angle brackets <>, but with functions, it's not that bad. The template parameters are mainly iterators and sometimes a function or a *functor*.

Assume you wanted to write an algorithm that calls a function for all adjacent pairs of a container range. That is, it does something similar to `std::for_each`, but for two adjacent elements of the container. Then the following applies:

- A container range is represented as usual by a pair of iterators.
- The two iterators are of the same type and a template parameter.
- The function takes two parameters of the same type.
- The function is a template parameter.

Let's call the algorithm `adjacent_pair`. Its signature can be derived from these considerations as follows:

```
template<typename It, Func>
void adjacent_pair(It begin, It end, Func func);
```

Thus, the function is flexible regarding the type of iterators and also the type of function. The implementation here is also not difficult.

```
// https://godbolt.org/z/n8ErWj7j6
template<typename It, typename Func>
void adjacent_pair(It begin, It end, Func func) {
    if(begin != end) {
        It prev = begin;    // first argument
        ++begin;           // second argument
        for(; begin != end; ++begin, ++prev) {
            func(*prev, *begin);
        }
    }
}
```

```

    }
}

```

Listing 25.19 A custom algorithm.

I use the `begin` parameter as the loop variable here, so I don't need to introduce another variable. What I do need, however, is a variable to remember the predecessor in my loop, so I can call `func` with it, as promised, with two adjacent elements.

The check at the beginning ensures that the call also works with an empty range. For a range with one element, it also works—no call to `func`—because it catches the loop's termination condition.

```

// https://godbolt.org/z/91jj8Toqs
#include <vector>
#include <iostream>
// ... adjacent_pair from above here ...
int main() {
    std::vector v{1,2,3,4};
    auto f = [](auto a, auto b) { std::cout << (a+b) << ' '; };
    adjacent_pair(v.begin(), v.end(), f); // 3 5 7
    std::cout << '\n';

    std::vector x{4,8};
    adjacent_pair(x.begin(), x.end(), f); // 12
    std::cout << '\n';

    std::vector w{4};
    adjacent_pair(w.begin(), w.end(), f); // nothing
    std::cout << '\n';

    std::vector<int> y{};
    adjacent_pair(y.begin(), y.end(), f); // nothing
    std::cout << '\n';
}

```

Listing 25.20 The use of `adjacent_pair`.

Some basic rules should be observed when implementing algorithms:

- Only require the properties of *forward iterators*. That means you should prefer the `++` increment. This way, you can use the new algorithm the most broadly.
- Compare iterators only with `equal ==` and not `equal !=`.
- When defining ranges using iterators, adhere to the rule that the *end* of a range is *exclusive*; end therefore points *behind* the last element.

- Consider edge cases, especially the empty range as a parameter.
- Prefer operations that do not generate temporary values—so instead of `it++` use `++it`.

Depending on the type of iterators you write your algorithm for, you determine which containers the algorithm can then work on. If you stick to forward iterators, the function works with all containers; with bidirectional iterators, it still works with `list`, but as soon as you require random access, you are limited to `vector`, `array`, and `deque`.

If *forward iterators* do not fit in your algorithm—because, for example, you want iterators to move forward and backward—then you prefer decrement `--` over addition `+` and subtraction `-` or over `advance()` and `distance()`. So you are still using bidirectional iterators, but not yet random-access iterators.

If you would use `<` instead of `!=` and `==` for iterator comparisons, for example, you would also need random-access iterators.

In [Chapter 26, Section 26.3](#), you will see that you can be even more general: *input iterators* cannot be stored like `Iter prev = begin` and reused later. If you want to write an algorithm for these, it is certainly possible. At the latest in this case, you should consider specialization via *traits* so that you do not lose performance with an overly ubiquitous implementation.

25.10 Writing Custom Views and Range Adapters

If you want to write your own range adapters for use in `operator|` chains, you need two things: an adapter class that implements your operation, and an instance of it, which is then the adapter itself. This is used to create *closure* objects. These objects are then instances where your adapter is “bound” to a specific range.

So you write a class `MyAdp` with `operator()` and then create an instance `myAdp` of it. If you now have a range `rng`, then `myAdp(rng)` is a range adapter closure object (or just a closure for short). The alternative way to create such a closure is to write `rng | myAdp`. You can then chain this closure with other `operator|` and ranges.

```
// https://godbolt.org/z/e54xzT9dW
#include <ranges>
#include <iostream>
#include <string_view>
#include <vector>

using namespace std::literals;
using namespace std;
namespace vs = std::views; namespace rs = std::ranges;
```

```

// Example 1
class Add_1: public rs::range_adapter_closure<Add_1> { // derive from helper class
public:
    template<rs::input_range R>
    constexpr auto operator()(R&& r) const {           // universal reference
        return forward<R>(r)                          // preserve universal reference
            | vs::transform([](auto i) {return i+1;}); // Your implementation
    }
};
Add_1 add_1{};                                         // Create range adapter

// Example 2
class Dna_to_rna: public rs::range_adapter_closure<Dna_to_rna> { // derive
public:
    template<rs::input_range R>
    constexpr auto operator()(R&& r) const {           // universal reference
        return forward<R>(r)                          // preserve universal reference
            | vs::transform([](char c)                 // Your implementation
            {
                switch(c) {
                    case 'T': return 'U';
                    case 't': return 'u';
                    default: return c;
                }
            });
    }
};
Dna_to_rna dna_to_rna{};                             // Create Range-Adapter
// Use examples
int main() {
    vector vec{1, 2, 3, 4, 5};
    for(auto i: vec | add_1)                            // use
        cout << i << ' ';
    cout << '\n';                                       // Output: 2 3 4 5 6
    auto telo_rep = "TTAGGGTTAGGGTTAGGGTTAGGGT"sv;
    for(auto c: telo_rep | dna_to_rna)                  // use
        cout << c;
    cout << '\n';                                       // Output: UUAGGGUUAGGGUUAGGGUUAGGGU
}

```

Listing 25.21 Custom range adapters for views.

Note the following things for your own adapter classes:

- Your operation must be wrapped in an adapter class, and it must have an `operator()` method that takes a range as a parameter.
- The range parameter should be a universal reference `&&` so that you can work with temp-values and variables.
- You will likely pass the range parameter along. To preserve the universal reference, you should use `forward<R>` for that.
- `forward` needs the concrete type `R`, which is why you cannot use an abbreviated function template with `auto`, but must write out the template. For this, use an appropriate range concept such as `input_range`.
- You should derive from the `range_adaptor_closure` base class, which was added in C++23. You may have noticed that you need to pass your closure class as a template parameter to it. This is the *curiously recurring template pattern* (CRTP), which helps the base class generate code for your adapter class.
- Use as much `constexpr` as possible.

Then create an instance of your adapter class for use in pipelines. In the example, `dna_to_rna` is an instance of `Dna_to_rna`.

Chapter 26

Good Code, 6th Dan: The Right Container for Each Task

Here I provide a decision aid on which container is right for which task. In the end, I will once again concretely bring together the application of containers and algorithms, because algorithms are part of solving tasks with containers.

26.1 All Containers Arranged by Aspects

It is possible to find comprehensive descriptions of the behavior of any given container in a multitude of places. This allows you to select the most appropriate container for a particular scenario “simply” by reading all the documentation.

Or you can read this section, where I do it the other way around: given a scenario or important aspect, and I tell you the strengths and weaknesses of the individual containers for it. Then you can choose the container that fits the given situation.

26.1.1 When Is a “vector” Not the Best Choice?

The first rule of thumb is this: use a `vector`. Only if that is not suitable should you use something else.

A `vector` has so many good properties that there must be a good reason to decide against it:

- **vector versus array**

If you don't want the elements to end up on the heap, but rather be managed on the stack like automatic variables, you can consider an `array`. `array` does not guarantee that the memory is always managed in the same place, but in practice it is very likely to be the stack. Alternatively, you can also use a `vector` with a special allocator.

- **vector versus deque**

If `push_back` and `pop_back` are not enough when inserting, because you also need `push_front` and `pop_front`, then use `deque`.

- **vector versus list**

If a `vector` grows dynamically, copying elements is expensive or impossible, and you don't know beforehand how many elements will come, then you can switch to `list`. It guarantees that once inserted, elements stay in place. A second reason could be

that you often need to insert in the middle. In both cases, you should be aware of the management overhead of `list`.

■ **vector versus set**

If all elements always need to be kept sorted, then use a `set`. However, read in the next section that it is worth considering whether you can split your work into a writing phase and a reading phase, because then `vector` and `sort` between the phases are the best solution.

■ **vector versus map**

If you need an association from a key to a target, a `map` is usually the best choice.

There are a few key tips to keep in mind when using `vector`.

You should add new elements to the back as often as possible. You do this with `push_back` or `emplace_back`. You should also remove elements from the back. The method for this is called `pop_back`.

If you already know or can estimate how many elements you will insert, you should request this amount with `reserve`. The automatic growth of `vector` is surprisingly efficient, but if you can avoid it, why not?

26.1.2 Always Sorted: “set”, “map”, “multiset”, and “multimap”

Before using `vector` (or `list`) to keep elements constantly sorted, you should prefer one of the ordered associative container types. An `insert` in the middle of a `vector` is really expensive. Because the elements in a `vector` are stored contiguously, in the case of an insertion that does not occur at the end, all elements must be shifted one position to the right. This also applies to `array` and `deque`, and although the latter does not guarantee completely contiguous storage for the elements, inserting in the middle is also sub-optimal.

If you decide on a constantly sorted `vector`, starting from C++23, `flat_set` is available. It is as easy to use as a `set` but stores its elements in a `vector`. This does not nullify the cost of an insertion in the middle; if you do not have too many elements, and they are also small, you can use `flat_set`.

Against keeping sorted with `list` or even `forward_list`, it argues that finding the insertion position takes a while. Even if a `list` is sorted, you usually have to traverse from the beginning to the desired element when searching. This is faster in a sorted `vector` or `array`. And the associative sorted containers are specifically made for this task.

26.1.3 In Memory Contiguously: “vector”, “array”

For most applications, it is a huge advantage if the elements are densely packed in memory—and also nicely in a contiguous manner. This is exactly what `vector` and `array` guarantee.

There are two main reasons for this:

- **Locality**

When you iterate through the elements of a `vector`, you usually start at the beginning and move toward the end in small or large jumps. The closer the memory accesses are within a unit of time, the better for the computer: memory is accessed in medium-sized blocks. For example, if you want to read only one byte, the CPU will still load a four-kilobyte page. The next time you read memory near the first byte, which is on the same page, it does not need to be loaded again. A huge time saver. The strategy of an implementation under this aspect is called locality behavior.

- **Shared access**

Wouldn't it be useful if you only needed to know a single memory address to process a million elements? This is possible with `vector` and `array`. If you know that your container holds a million elements, a single call to `data()` is enough, and you have all the information you need. From that point on, the elements lie neatly next to each other. You can use this, for example, to pass all elements together to a C function. Many C functions take a raw pointer plus a count to receive multiple elements. This is not only useful for interacting with C: writing a single large block of data from memory to disk is much faster than multiple small ones, and some C++ functions also take advantage of this.

The “deque” Does Not Guarantee Contiguous Storage

Regarding the locality of elements, `deque` is as good as `vector`. The elements are very likely stored directly next to each other and therefore require very little additional management overhead. When you iterate through them, it is memory-friendly in one direction.

But not *all* elements are guaranteed to be stored next to each other. There can be several large chunks. This means there is no way to access the single memory area of the elements with `data()`, as is the case with `array` and `vector`. Therefore, this application is unfortunately ruled out for `deque`.

26.1.4 Cheap Insertion: “list”

Inserting a single element somewhere in the middle of a `list` (and `forward_list`) always costs a constant amount of time—called $O(1)$ (see [Chapter 24, Section 24.1.4](#)). No other container can exhibit this property:

- `vector` (and `deque`) can insert at the end (and beginning) in $O(1)$, but require linear time in the middle—called $O(n)$.
- `map` and `set` (and the `multi...` variants) guarantee $O(\log n)$, and the position is determined by the sorting.

- `unordered_map` and `unordered_set` usually insert in $O(1)$, but in the worst case, which is hard to predict, in $O(n)$. The position of the insertion also cannot be determined and may change with further insertions.

26.1.5 Low Memory Overhead: “vector”, “array”

If you want to store a million small elements like `int`, it can't be done more compactly than in `vector` or `array`. You need exactly zero bytes of additional memory per element. That's not much and can't be beaten by any other container—except perhaps `deque`, which requires a little additional memory for block management, but very little.¹

All other containers require more additional memory per element. Rarely can this be reduced with tricks:

- `list` stores two additional pointers per element. On a current Linux system, an `int` requires four bytes and a pointer requires eight bytes of memory. For one million `int` elements, you need four megabytes of memory with a `vector`. In a `list`, you would occupy 20 megabytes.
- `forward_list` is specifically designed for low memory overhead. If `vector` (without any overhead) does not work, but the smallest possible additional footprint per element is crucial, then `forward_list` is preferred. However, keep in mind that you will have to deal with its specialized interface, which I advise against unless necessary.
- The ordered associative containers store their elements in a tree. Typically, this means that three additional pointers are required per element (although optimizations exist).
- The unordered associative containers maintain a hash table and a set of buckets with the actually inserted elements. The hash table holds pointers and, when well-filled, has about as much space for pointers as you have elements in the container.² The buckets are usually implemented as `forward_list`. All in all, you need an estimated two additional pointers of space per element in your container.

Besides the additional memory required per element, there is another factor you need to consider: It is much worse to request 1,000 bytes 1,000 times than to request 1,000,000 bytes once. This is because the system also requires some memory for each requested memory block. It gets even worse if you also release blocks in between. Then it depends on how well your system can reuse memory—and that is an arbitrarily difficult problem.

What can happen is that the memory *fragments*. The bad thing is that it happens without you noticing, hidden from you. Your program has no way to detect fragmentation.

1 There are implementations that do not require additional memory per element.

2 I say *space for pointers* here instead of *number of pointers* because some slots in the hash table remain unoccupied. So no pointer is stored there, but space is still consumed.

You only notice it when you request memory that is theoretically available but can no longer be provided to you. Yes, there are solutions for this, but they are situation-dependent. One of the possible solutions is this: use a `vector`.

Noteworthy Special Case: “`vector<bool>`”

A `vector<bool>` is optimized for low memory usage. It packs the elements tightly—that is, eight `bool`s per byte. [Chapter 24](#), [Table 24.12](#) presents the different alternatives for storing many `bool` values.

26.1.6 Size Dynamic: All Except “array”

`array<int,5>` means space for five. Not four, not six, but five. You can neither remove elements from the `array` nor add any. All other containers grow with their tasks: they start empty and with possibly preallocated memory and grow dynamically as you add more elements. In most cases, you don't have to worry about the additional overhead when growing.

But there are exceptions. Here are the facts about the growth of containers in the C++ standard library:

- `array` cannot grow.
- `list` and `forward_list` manage singly linked elements. They behave optimally when growing: the elements are inserted individually and independently into the data structure. However, many small memory allocations can cause memory fragmentation.
- `vector` can grow well, but it is not optimal. Occasionally, `vector` has to double its size and temporarily occupies three times the memory than actually needed for the elements, but immediately releases one-third of it. However, the memory does not fragment, even if you store many small elements.
- The ordered associative containers link elements individually in a tree-like structure. This is also good for growth. However, parts of the tree need to be rearranged when inserting, which involves some effort.
- The unordered associative containers also store the elements individually linked, which is good for adding elements one by one. However, the hash table occasionally needs to be resized. To estimate the effort of resizing, assume the hash table is a `vector` with approximately as many pointers as the container has elements.
- The `deque` has the most degrees of freedom when it comes to implementation. It grows well, at worst like a `vector`. However, it is more likely that memory is requested in larger chunks and not as much temporary storage is needed.

That's all there is to say about growth. But what about shrinking?

- You can't remove anything from an `array`.
- `list` and `forward_list` as well as all associative containers manage their elements individually, and removing an element also frees its memory. Only in the case of unordered associative containers might the hash table not be affected by shrinking.
- `vector` does not shrink when you remove elements. The maximum space that a vector once required it will continue to require, even if you delete elements. You can explicitly call `shrink_to_fit`, which releases (parts of) the excess memory. The implementation may do this, but it doesn't have to. However, all implementations known to me support `shrink_to_fit` to the next larger power of two. Note that you should not call `shrink_to_fit` after every removal.
- `deque` can be implemented like `vector` and therefore has `shrink_to_fit`, but it can also be implemented with blocks and automatically behave better regarding removal.

26.2 Recipes for Containers

After the general guide to container selection, I will show you some specific recipes for handling containers. I have chosen problems that frequently occur in practice:

- Is a `set` really the right container for a sorted collection?
- How do I output the contents of a container?
- Can I really not add an element to an `array`?

Answering these questions in the following sections also helps in understanding some fundamental aspects of C++ and the standard library: the linear memory layout of `vector`, the interaction between streams and containers, and the moving of values.

26.2.1 Two Phases? “vector” as a Good “set” Replacement

Note that whenever your use of `set` splits into two distinct phases for filling and reading, you should prefer using a `vector`. A `set` keeps its elements sorted at all times and incurs some overhead with *every* insertion. Often, it is better to first dump all elements into a `vector` using `push_back()` and then sort them in one go with `sort()` from the header `<algorithm>`.

```
// https://godbolt.org/z/T4d4jdqox
#include <vector>
#include <iostream>
#include <algorithm>
using std::vector; using std::ostream; using std::cout;
int main() {
```

```

vector<int> data{};
data.reserve(400);           // Space for 400 elements
// Phase 1: fill
for(int idx = 1; idx <= 20; ++idx) {
    for(int val = 0; val < 20; ++val) {
        data.push_back(val % idx);    // something between 0 and 19
    }
}
cout << data.size() << '\n';        // 400 elements between 0 and 19
// Post-processing Phase 1: create set-equivalent
std::sort(data.begin(), data.end()); // preparation for unique
auto wo = std::unique(data.begin(), data.end()); // duplicates to the end
data.erase(wo, data.end());          // remove duplicates
data.shrink_to_fit();
cout << data.size() << '\n';        // only 20 elements left
// Phase 2: use
for(auto &e:data)
    cout << e << ' ';              // Output: 0 1 2 .. 18 19
cout << '\n';
auto it = std::lower_bound(data.begin(), data.end(), 16); // search value
if(it!=data.end() && *it == 16)
    cout << "found!\n";
if(std::binary_search(data.begin(), data.end(), 7))    // yes or no
    cout << "also found!\n";
}

```

Listing 26.1 With a “vector”, you can simulate a “set”.

If you then iterate over the elements of the vector, you get the same result as with a multiset. To simulate a set, follow the call to `sort()` with a call to `unique()` from the header `<algorithm>` and then remove duplicates with a call to `erase()`.

Then you can use the vector almost as if it were a set. The elements are sorted, which means you can read them out in the same order. Instead of using `find()` to search for elements, you now use, for example, the functions `lower_bound()` or `binary_search()` from `<algorithm>`. This is done using *binary search* and is therefore just as fast as set—probably even faster as vector has better *locality behavior*, meaning the data is stored in a more CPU-friendly manner in memory.

Binary Search

This is a standard procedure to ensure that a desired element can be found in a sorted sequence of elements with a few steps. In this process, the middle element of the sequence is successively selected and then searched either to the right or left, depending on whether the middle element is greater or smaller than the desired element. This

guarantees that the desired element is found after a maximum of $\log n$ steps or it is known that it is not in the list.

The disadvantage of this procedure is that you keep more elements in the vector during the insertion phase than in a set. This means that vector loses its speed advantage over set if you expect many duplicates. When exactly this happens depends on many factors. As a rule of thumb, I would recommend that using vector as a set replacement is worthwhile if you expect more than about 1,000,000 elements and more than half of them remain unique. For fewer elements, the additional effort in implementation is hardly worthwhile, and for fewer unique elements (i.e., more duplicates), set saves memory.

26.2.2 Output the Contents of a Container to a Stream

This recipe is needed repeatedly: for example, for debugging or during unit tests. Of course, there are a thousand ways to output the elements of a container to a stream: from a simple loop to a ranged for loop to a specialized overload of the operator<<.

The most elegant way, however, is to copy the contents of the container to the stream—for this, `std::ranges::copy()` from `<algorithm>` is used, or `std::copy` before C++20. Then, all that is needed is the `ostream_iterator` from `<iterator>` as an adapter, and the output is a one-liner:

```
// https://godbolt.org/z/fGKsfs1Y4
#include <iostream> // cout
#include <algorithm> // copy
#include <iterator> // ostream_iterator
#include <vector>
int main() {
    std::vector<char> path{};
    for (char ch = 'a'; ch <= 'z'; ++ch) {
        path.push_back(ch);
    }
    std::ranges::copy(path, // here everything, but it also works with other ranges
        std::ostream_iterator<char>(std::cout, " ") // copy to cout, separator " "
    );
}
```

26.2.3 So “array” Is Not That Static

After reading the chapter on containers, you might think that array always has a fixed size. While this is true, you don't have to completely discard array if you need one more

or one less element. It is generally quite simple to add an element to an array, merge two together, or shrink it.

Almost. You cannot turn an `array<int,1000>` into an `array<int,1001>` without copying the integers once. But for (large) elements like perhaps `Picture`, which you can move, you can indeed turn an `array<Picture,50>` into an `array<Picture,51>` without copying a lot of data.

It is quite easy to enlarge any `array<T,n>` by one element.

```
// https://godbolt.org/z/Yde6azqoc
#include <iostream>
#include <array>
#include <vector>
#include <string>
using std::array; using std::move; using std::forward;

// == enlarging array ==
template<typename T, size_t S, std::size_t... Idx>
constexpr array<T, S+1>
help_append(array<T, S>&& data, T&& elem, std::index_sequence<Idx...>) {
    return { std::get<Idx>(forward<array<T, S>>(data))..., forward<T>(elem) };
}
template<typename T, size_t S>
constexpr auto
append(array<T, S> data, T elem) {
    return help_append(move(data), move(elem),
        std::make_index_sequence<S>{});
}

// == Example ==
class Picture {                // rule of zero; movable
    std::vector<char> data_; // lots of data
    std::string name_;
public:
    explicit Picture(const std::string& name) : data_(1000,0), name_{name}
        { /* ... load picture here ... */ }
    auto name() const { return name_; }
};

int main() {
    // before
    array<Picture,3> pics{Picture{"Mona"}, Picture{"Scream"}, Picture{"Vincent"}};
    std::cout << pics[0].name() << '\n'; // Output: Mona
    // enlarge
    Picture new_pic { "Clocks" };
    auto more = append(move(pics), move(new_pic));
```

```
// after
std::cout << pics[0].name() << '\n'; // Output:
std::cout << more[0].name() << '\n'; // Output: Mona
std::cout << more[3].name() << '\n'; // Output: Clocks
}
```

Listing 26.2 You get back an array enlarged by one element.

First, let me explain what happened in `main()`. In the middle, you see a call to `append`. It is important here that you pass both arguments with `move`; otherwise, the `append` function could not move the data out of its parameters. As a reminder: with `std::move`, you convert an lvalue reference into an rvalue reference—a temp-value—and thereby tell the compiler that it is allowed to “steal” the data from the argument. If you omit this, you still get a larger array back, but the original `pics` and `new_pic` parameters would remain unchanged. However, the data was moved out of them, which you can see by the fact that `pics[0].name()` is empty. Also, `pics[0].data_` is now empty.

You still have three `Picture` elements in `pics`, but they are all empty and are now in the first three slots of the array `<Picture, 4>` named `more`.

How did that happen? The key lies in the helper function `help_append` and in `index_sequence`. First, `append` calls the helper function:

```
return help_append(move(data), move(elem), std::make_index_sequence<S>{});
```

I'll write this in pseudocode, where the functionality of `make_index_sequence` becomes a bit clearer. The call effectively results in the following:

```
return help_append(move(pics), move(new), tag<0,1,2>);
```

With the template function `make_index_sequence<S>`, you generate a tag type at compile time that somehow contains a sequence of numbers—specifically, the numbers needed to directly access the individual elements from `pics`.

The helper function now constructs a new array:

```
return { std::get<Idx>(forward<array<T, S>>(data))..., forward<T>(elem) };
```

The braces `{` and `}` form an initializer list. And because the return type is given as `array<T, S+1>`, it specifically initializes an `array<Picture, 3+1>`.

The `get` is central here. The ellipses `...` refer to the passed `index_sequence`, which is `<0,1,2>` (pseudocode). The compiler expands this to `get<0>(d)`, `get<1>(d)`, `get<2>(d)`. Finally, `elem` is added to the initializer list—in this case, `new_pic`.

But not quite: to actually move, each `d` is actually `forward(data)` and `new_pic` is actually `forward(new_pic)` (with the respective template arguments). Using `forward` and the *universal reference* `&&` in the parameter list, the function works for both values to copy and temp-values to move.

So if you have movable values in an array, you can still manipulate the array cost effectively.

Consider the following implementation using `operator[]` for comparison.

```
// https://godbolt.org/z/MMP16Mj3W
#include <iostream>
#include <array>
#include <string>
using std::array;
template<typename T, size_t S>
auto append(const array<T, S>& data, T elem) {
    array<T, S+1> result {};
    for(auto i=0u; i < data.size(); ++i)
        result[i] = data[i];
    result[S] =elem;
    return result;
}

int main() {
    // before
    array pics { 3, 4, 5 };
    std::cout << pics[0] << '\n'; // Output: 3
    // enlarge
    auto more = append(pics, 77);
    // after
    std::cout << more[3] << '\n'; // Output: 77
}
```

Listing 26.3 “append” at runtime.

Here is an implementation of `append` that uses `operator[]` instead of `get`. For better clarity, I have removed everything related to moving, but that makes no difference for what I want to explain.

The main difference between `get<>` and `operator[]` is, of course, that the index for access with `get<>` is decided at compile time, whereas with `operator[]` it is decided at runtime. And because of this, you can use `get<>` together with the ellipsis `...` which the compiler expands to index accesses at translation time. At runtime, you don't have this feature, so you have to resort to a `for` loop.

There are only small speed advantages that you get with `get<>` instead of `operator[]`. It is more likely that the compiler can optimize something better. Because with the `for` loop, `append` is now too complicated for `constexpr`. And if there's one thing the compiler likes for its optimization attempts, it's `constexpr`.

26.3 Implementing Algorithms That Are Specialized Depending on the Container

If you want to write algorithms that behave differently depending on the type of range (or container) they operate on, starting with C++20, you can use `constexpr if` and the well-thought-out hierarchy of ranges concepts to write the appropriate implementation in the different `if constexpr` branches.

Within a single function, you can decide which algorithm you want to use for which range. All of this happens at compile time, as shown in [Listing 26.4](#).

```
// https://godbolt.org/z/rqz1s6h11
#include <iostream>
#include <vector>
#include <list>
#include <ranges>
namespace rs = std::ranges;

template<rs::range R>
void alg(R&& range) {
    if constexpr(rs::random_access_range<R>)
        std::cout << "random access.\n";
    else if constexpr(rs::bidirectional_range<R>)
        std::cout << "bidirectional, but not random access\n";
    else static_assert(false, "unsupported range type");
}

int main() {
    std::vector<int> vec {};           // vector is random access
    alg(vec);
    std::list<int> lst;               // list is only bidirectional
    alg(lst);
    std::istreambuf_iterator<char> i1{std::cin}, i2{}; // not even bidirectional
    auto fwd = rs::subrange{i1, i2};
    alg(fwd);                        // ✗ Error: no matching implementation
}
```

Listing 26.4 Different implementations of an algorithm, distinguished by concepts.

Before C++20 with its ranges, concepts, and `constexpr if`, this was not so easily possible. The way then led through the *iterator tags*, with which you can offer multiple overloads of a function. The code would thereby become much more confusing.

With `constexpr if`, you have more flexibility than would be the case with overloads. Here, `random_access_range` first checks the special case, then `bidirectional_range` the more general one.

Chapter 27

Streams, Files, and Formatting

Chapter Telegram

- **Stream**
Model of a class for input and output.
- **istream, ostream, and iostream**
Base classes for input and output streams.
- **operator<< and operator>>**
Important operators for output and input.
- **cin, cout, cerr and clog**
Standard streams for normal input, output, and error output.
- **<filesystem>**
Header with utilities for working with files in the filesystem.
- **format from <format>**
Function for flexible formatting from C++20.
- **print and println from <print>**
Output using `format` from C++23.

Handling files and input/output in C++ is realized through a special and extensible stream model. This chapter will explain this fundamental stream model of C++ in more detail.

Streams sometimes—but not always—refer to files. Handling names and other attributes of files on the hard drive is described at the end of this chapter.

Sometimes, using streams feels a bit cumbersome, especially when it comes to changing formatting. From C++20, there is a new formatting library in the `<format>` header, which allows convenient formatting attributes and is extensible. It is entirely independent of streams. Direct formatted output will be possible from C++23 with `print` and `println` from the `<print>` header. We will discuss both at the end of this chapter.

27.1 Input and Output Concept with Streams

On the following pages, you will very often read the term *stream* or *stream object*. `cout` or `cin` are examples of such (globally predefined) stream objects. A stream is a flow of

data that moves in a specific direction. Each stream object has its own properties, which are defined by the corresponding stream class.

The entire principle is based on a hierarchy of class templates, with the `ios_base` class at the top of the class hierarchy, from which all other stream classes are derived. [Figure 27.1](#) shows a simplified representation of the input/output class hierarchy.

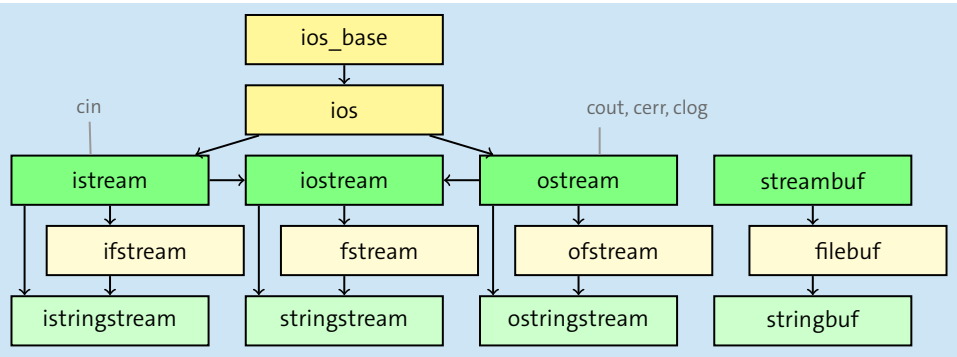


Figure 27.1 Simplified representation of the stream hierarchy.

The I/O stream library with all its facets is extremely extensive, so not all details of the library can be covered here.

Specializations

In the representation in [Figure 27.1](#), the specializations for a base type have already been shown. Apart from `ios_base`, the `basic_` prefix would appear before each class template here. A specialization thus looks as follows:

```
using ostream = basic_ostream<char>;
```

This is a specialization for the `char` data type from the `basic_ostream` class template. Such specializations can also be used for other character types like `wchar_t`, `char16_t`, and so on.

27.2 Global, Predefined Standard Streams

The I/O stream library provides predefined and global stream objects that you have used practically since your first C++ program without knowing what they are. Of course, we are talking about the standard stream objects `cin`, `cout`, `cerr`, and `clog` from `std`.

For output, the `<ostream>` header file contains the `cout` (standard output), `cerr` (standard error output, unbuffered), and `clog` (standard error output, buffered) stream

objects. For input, the `<istream>` header file contains the `cin` stream object, which can be used for typical user inputs (e.g., from the keyboard).

27.2.1 Stream Operators `<<` and `>>`

Basically, a data stream is initially nothing more than an unreadable sequence of bytes, which is formatted for output by specializing `operator<<` and for input by specializing `operator>>`.

In the `<ostream>` header file, you will find `operator<<` overloaded multiple times for all built-in types:

```
ostream& operator<<(bool val);
ostream& operator<<(short val);
ostream& operator<<(unsigned short val);
ostream& operator<<(int val);
// ...
```

The same applies to `operator>>` in the `<istream>` header file:

```
ostream& operator>>(bool val);
ostream& operator>>(short val);
ostream& operator>>(unsigned short val);
ostream& operator>>(int val);
// ...
```

By returning the stream as a reference with the `operator<<` and `operator>>` operators, it becomes possible to use the operators multiple times in a row, as in the following listing.

```
// https://godbolt.org/z/WaEGdnWvq
#include <iostream> // cin, cout
int main() {
    int val1, val2;
    std::cout << "Please enter 2 int values: ";
    std::cin >> val1 >> val2;
    std::cout << val1 << " : " << val2 << std::endl;
}
```

Listing 27.1 Returning the stream by reference allows chaining.

Using Custom Types

Many standard classes like `string` also have the `operator<<` and `operator>>` operators implemented. For custom classes, you need to implement these yourself if you want `<<` and/or `>>` to work with streams.

The standard input/output operators support almost all fundamental types in C++. Exceptions are `void` and `nullptr_t`. Of course, input and output do not occur arbitrarily. There are also a few things you need to consider:

- **Numeric types**

When reading a numeric value, the input must start with at least one digit. If you specify the sign with `+` or `-`, it must only appear before the first digit. However, there must be no space between the sign and the digit. Leading whitespace characters are ignored. In case of an erroneous input, the numeric value is set to 0 and `std::failbit` is set.

- **char***

The use of `char*` for input was already very critical in C. Input is read until the first whitespace or until *end of file* (EOF; more on this later). Leading whitespace is skipped, which can be changed with the `skipws` manipulator. The problem with reading `char*` is that it can read beyond the buffer area (buffer overflow)—for example, `char cval[20]; std::cin >> cval`. If you read in more than 20 characters here, `cin` will dutifully do so and write into an undefined memory area. Therefore, with old C-strings (if you absolutely must use them), you must always ensure that the maximum length is not exceeded. For this, the `setw` manipulator from the `<iomanip>` header is suitable—for example, `cin >> setw(20) >> cval`.

- **bool**

The output of `bool` is converted to 0 for false and to 1 for true. For input, the same rules as for numeric values apply initially. When entering valid numeric values other than 0, it is converted to 1 (= true). When entering 0, 0 (for false) is used. In case of an erroneous input, the value is also set to 0 and `std::failbit`.

27.3 Methods for Stream Input and Output

In addition to the usual `operator<<` and `operator>>` for streams, you can also use methods provided by the I/O stream library. However, unlike the `<<` and `>>` operators, these methods work with raw bytes and are therefore unformatted. The methods for standard input/output use the `streamsize` type from `<ios>`, which is usually `size_t`. This means, for example, that leading whitespace is not ignored during input.

27.3.1 Methods for Unformatted Output

For raw and unformatted output, the I/O stream library provides the `put()` and `write()` methods from the `<ostream>` header file. The `put()` method outputs individual characters, and `write()` outputs a specified number of characters. The following code snippet demonstrates these methods briefly:

```

std::string sdata("raw data");
for(auto c : sdata)
    std::cout.put(c);
std::cout << 10 ;           // = 10
std::cout.put(10);          // = Newline '\n'
char cdata[] = {"raw data"};
std::cout.write(cdata, 4); // = raw
std::cout.put(10);          // = Newline '\n'
std::cout.flush();

```

With `put(c)`, you output the individual characters of the `sdata` string unformatted on the screen. With `<< 10` and `put(10)`, you can clearly see the comparison between formatted and unformatted output. While `std::cout << 10` outputs the formatted value 10, `std::cout.put(10)` outputs the unformatted (usually ASCII) value on the screen. A look at an ASCII table shows that the value 10 stands for the newline character.

Afterward, the unformatted output with `write()` is demonstrated. It simply outputs a specified number of characters (here, four) from `cdata` unformatted to the screen. Here, of course, there is again a risk because more characters can be specified than are actually available. At the end, we also force all data still in the stream's buffer to be output immediately with `flush()`.

Positioning Stream

To position streams, you can use the `tellp()` and `seekp()` methods, which are also included in `<ostream>`. These methods only become really interesting when you work with files. You can learn more about this in [Section 27.6](#).

The following is an overview of the methods for unformatted output in `<ostream>`:

- **`ostream& put(char c);`**
Writes the `c` argument to the stream.
- **`ostream& write(char* s, streamsize n);`**
Writes `n` characters from `s` to the stream. The caller must ensure that the value of `n` is not greater than the number of characters in `s`. The termination character for C strings `'\0'` does not end the output of `write()`.
- **`ostream& flush();`**
Flushes the stream buffer immediately. All characters in the buffer that have not yet been written must now be output.

All methods return the `ostream` stream. Whether the output was successful can and should be checked using the functions from the `<ios>` header file ([Section 27.4](#)).

27.3.2 Methods for Unformatted Input

When reading raw data with methods, unlike formatted reading with `>>`, leading whitespace characters are not simply skipped and ignored. For unformatted input, the header file `<istream>` offers several methods, some of which are overloaded multiple times.

The following is an overview of the methods for unformatted input in `<istream>`:

- **`istream& get(char& c);`**
Extracts a single character from the stream into `c`.
- **`istream& get(char* s, streamsize n);`**
This reads $n-1$ characters from the stream and stores them as a C string in `s`. If fewer than $n-1$ characters are used, reading continues until the first newline character `'\n'`. The newline character is not stored in `s`. The null terminator `'\0'` is automatically added at the end.
- **`istream& get(char* s, streamsize n, char del);`**
This reads $n-1$ characters from the stream and stores them as a C string in `s`. If fewer than $n-1$ characters are used, it reads up to the first delimiter character contained in `del`. The `del` character is not stored in `s`. The termination character `'\0'` is automatically appended at the end.
- **`istream& getline(char* s, streamsize n[, char del]);`**
The main difference from the corresponding `get()` versions is that the `failbit` error flag is set if more than $n-1$ characters are read.
- **`istream& read(char* s, streamsize n);`**
This extracts n (not $n-1$) characters from the stream and stores them in C string `s`. Unlike the `get` and `getline` versions, the `read` sequence is not terminated with the null character `'\0'`—a classic function for copying entire blocks of data.

For all versions where you extract $n-1$ characters from the stream into the address of `char* s`, you must ensure that there is enough memory space available in `s`. All methods return the stream `istream`. Whether the input was successful can/should be checked using the mechanisms available in the `<ios>` header file ([Section 27.4](#)). Extracting characters from a stream is terminated in all input methods when EOF is read.

End of File

EOF is often defined with the value `-1` and is used to indicate the end of a data stream or file. In an interactive shell, you can also generate EOF with a key combination. In classic Unix shells, you can reproduce this with `[Ctrl] + [D]`. In the Windows command prompt, EOF is generated using `[Ctrl] + [Z]`.

The following simple listing demonstrates some of the methods for unformatted input.

```
// https://godbolt.org/z/MaEzcccTG
#include <iostream>
using std::cout; using std::cin; using std::endl;
int main() {
    const unsigned int MAX = 10;
    char buffer[MAX] = {0};
    cout << "Input getline : ";
    cin.getline(buffer, MAX);
    cout << std::cin.gcount()
        << " characters were read\n";
    for(auto c : buffer) {
        if(c && c!='\0') cout.put(c);
    }
    cin.ignore(MAX, '\n');
    cout << "\nMake input (end with .) : ";
    char ch=0;
    while(cin.get(ch)) {
        if(ch == '.') break;
        cout.put(ch);
    }
    cout << "Input ended" << endl;
}
```

Listing 27.2 Unformatted input from streams.

First, I extract up to MAX characters from the `cin` stream and store them in `buffer`. The number of characters actually extracted from the stream is determined using the `gcount()` method. The `gcount()` method can also be used for all other methods of unformatted input.

Then I output the individual characters to the screen using `put()`.

In this example, I use the `ignore()` method to skip up to MAX characters from the input stream until the `'\n'` character and remove `'\n'` from the buffer. Instead of `'\n'`, another character can also be used here. In the example, I use this line to pull the new-line character out of the buffer so that it is not used in the next read operation and can thus be simply skipped.

This “trick” does not work everywhere. On other systems (such as the Mac), you can insert the following lines:

```
cin.clear();
cin.ignore(cin.rdbuf()->in_avail());
cin.get();
```

The `while` loop comes from the classic Unix framework, which simply reads all characters from the `cin` stream unformatted and outputs them to the `cout` stream. Reading continues until a period is entered or EOF is reached.

The program during execution looks like this:

```
Input getline: Hello getline
9 characters were read
Hello get
Make an input (end with .): Hello get
Hello get
Another line
Another line
We end with a period.
We end with a period
Input ended
```

std::string and getline()
In the `<string>` header file, you will also find a `getline()` method that reads a line from a stream into a `std::string`.

27.4 Error Handling and Stream States

For streams, it is often not enough to simply check whether something works (`true`) or does not work (`false`). Because various types of errors can occur, the I/O stream library provides mechanisms to check the state of the stream. For this, you will find a few constants with `iostate` from `ios_base` (see [Table 27.1](#)).

The constants are defined in `ios_base` as public members and can be accessed either directly via `ios_base` (e.g., `ios_base::goodbit`) or through classes derived from `ios_base` or created objects (e.g., `ios::goodbit` or `cout.goodbit`).

Constant	Meaning
<code>goodbit</code>	No problems with the stream. All other bits are not set.
<code>eofbit</code>	EOF is set. The end of a data stream has been reached.
<code>failbit</code>	Error in an input/output function. An input/output operation could not be successfully executed.
<code>badbit</code>	Severe error. The stream is in an undefined state.

Table 27.1 Constants in `iostate`.

27.4.1 Methods for Handling Stream Errors

To query the individual states, C++ offers you various methods:

- **bool good()**
Returns true if everything is okay (goodbit is set).
- **bool eof()**
Returns true if the file stream has ended (eofbit is set).
- **bool fail()**
Returns true if an error has occurred (failbit or badbit are set).
- **bool bad()**
Returns true if a serious error has occurred (badbit is set).
- **void clear()**
Resets the state of iostate. Status is set to goodbit.
- **void clear(iostate f)**
Resets the state of iostate and sets the status to f.
- **void setstate(iostate f)**
The current state is combined with f using a bitwise OR.
- **iostate rdstate()**
Returns the currently set status.

eofbit and failbit Set

Usually, failbit is also set when eofbit is set. The reason is that the eofbit state is only set after an attempt is made to perform another action past the EOF. This also means that the next operation fails, and therefore failbit is set as well.

```
// https://godbolt.org/z/exKMv8646
#include <fstream>
#include <iostream>
using std::cout; using std::cin; using std::ofstream;
void checkIOstate(std::ios& stream) {
    if( stream.good() ) {
        cout << "All good\n";
    } else if( stream.bad() ) {
        cout << "Fatal error\n";
    } else if( stream.fail() ) {
        cout << "I/O error\n";
        if( stream.eof() ) {
            cout << "End of stream reached\n";
        }
    }
    stream.clear();
}
```



```
int main() {
    int val=0;
    cout << "Enter value: ";
    cin >> val;
    checkIOstate( cin );
    std::ifstream file;
    file.open("nonexistent.txt");
    checkIOstate(file);
    std::fstream fstr;
    fstr.open("newFile.txt",
        ofstream::out | ofstream::in
        | ofstream::binary | ofstream::trunc);
    fstr << "Text in the file\n";
    fstr.seekp(std::ios_base::beg);
    char ch;
    while( fstr.good() ) {
        fstr.get(ch);
        if(fstr.good()) cout.put(ch);
    }
    checkIOstate(fstr);
}
```

Listing 27.3 State checks for streams.

Pass any stream to the `checkIOstate()` function to check its state. I check the individual methods to clarify the state. At the end, I reset the state with `clear()` back to goodbit.

In the section starting from `val=0`, you will be prompted to enter a value. Correctly, you must enter an integer here. With `checkIOstate`, I then check the `cin` stream. If the input is correct, goodbit is set, and `good()` is true. For an incorrect input like `blablubb`, `fail()` true.

For `ifstream` `file`, I check a file stream `std::ifstream`, because with it I can open a file for reading. In the example, it is assumed that the file does not exist, which is why `fail()` is true here as well.

And finally, starting from `fstream` `fstr`, I demonstrate EOF. Between `open` and `seekp`, I create a file, write something into it, and reset the data stream to the beginning. In the `while` loop, I read characters from the `fstr` stream as long as the state of `fstr` is equal to `good()` (goodbit set). If the state is no longer `good()`, either the end of the data stream has been reached (eofbit is set) or an error has occurred (failbit is set). As already mentioned, failbit is usually also set when eofbit is set.

Exception `ios_base::failure`

In case of errors in the I/O stream library, the `ios_base::failure` exception can also be thrown (if `badbit` is set). However, this depends on the implementation of the compiler manufacturer. Nevertheless, it is possible to throw and catch this exception yourself using `throw`.

Checking Streams with “bool”

The state of the stream can also be tested with `bool` and the `!` operator. Both have been implemented as methods in `ios_base` (`operator bool()` and `operator!()`). Both methods return whether an error has occurred (`failbit` or `badbit` set).

For example, the operator `bool` is used as shown in the next listing.

```
// https://godbolt.org/z/Mr6E3Ez48
#include <iostream>
int main() {
    unsigned int val;
    std::cout << "Enter value: ";
    std::cin >> val;
    if( std::cin ) { // operator bool()
        /* ... */ // Input correct
    } else {
        std::cout << "Error with std::cin\n"; // Error with input
    }
}
```

Listing 27.4 Operator “bool” of streams.

In `if( std::cin )...`, the stream, because it is used in an `if`, is automatically converted to `bool`. The operator `bool()` is defined in the class specifically for such purposes. You can just as easily apply the `!` operator to streams.

27.5 Manipulating and Formatting Streams

How characters are input or output in a stream is controlled by so-called flags, which are defined in the `ios_base` base class. You can think of flags as an array of bits, where the values can be either 1 or 0. Individual bits, of course, do not make a value, so the bits are stored in a variable. Depending on the set bits, the value of the variable changes, and each value has a special meaning. All flags have a default setting, where, for example, without further specification, all integral values are output in decimal (`dec`) notation.

You can directly change the format flags by using methods like `setf()`, `unsetf()`, or `flags()` from the `ios_base` class. However, a better alternative has been available for a long time (specifically) with manipulators. With these, you can insert and call all flags directly as a function in an input or output stream. In practice, the use of manipulators is therefore usually preferable to direct access to the format flags with methods. Therefore, manipulators will be discussed first. The methods by which you can directly change the format flags will be briefly introduced in [Section 27.5.4](#).

27.5.1 Manipulators

Manipulators for streams are objects that you can insert with the standard input/output operators `<<` and `>>`. You are already familiar with the manipulator `endl`. This adds a newline character to the buffer to output a new line. The basic manipulators are listed in `<istream>` and `<ostream>` in [Table 27.2](#).

Manipulator	Meaning
<code>endl</code>	Outputs a new line and flushes the output buffer.
<code>ends</code>	Adds the null terminator character (' <code>\0</code> ') to the buffer.
<code>flush</code>	Flushes the stream's output buffer.
<code>ws</code>	Ignores leading whitespace when reading input.

Table 27.2 Basic manipulators in `<istream>` and `<ostream>`.

Independent Manipulators in “`<ios>`”

You can find a considerable number of additional manipulators in the header file `<ios>`. These are manipulators to enable or disable certain states or to format the data during output. I summarized the manipulators in [Chapter 6](#), [Table 6.3](#).

Since C++20, you can often use `format()` instead of the manipulators. This allows you to convert built-in and custom types into a string. You can find the description in [Section 27.10](#).

It is easy to enable or disable states. Once (de)activated in a particular stream, the state remains until you call the opposite manipulator again. Consider the following simple example.

```
// https://godbolt.org/z/s1csa51z9
#include <iostream>
#include <iomanip>
using std::cin; using std::cout; using std::endl;
void f(bool b) {
    cout << b << endl;           // Output: true
}
```

```

int main () {
    bool b=true;
    cout << std::boolalpha << b << endl; // Output: true
    b=false;
    cout << b << endl; // Output: false
    f(true);
    cout << std::noboolalpha << b << endl; // Output: 0
    b=true;
    cout << b << endl; // Output: 1
}

```

Listing 27.5 Outputting a Boolean as text or a number.

First, I push `std::boolalpha` into the `cout` stream to output `true` or `false` from Boolean variables literally, which the output in the same line also immediately proves. The `cout << b` afterwards also outputs the value of the Boolean variable literally. From this, you can see that the setting, once activated, remains active.

The output in the function `f()` shows that the setting is globally applicable. After the function call, I deactivate the setting again by sending `std::noboolalpha` into the stream. The output of the now numeric values of 0 and 1 instead of `false` and `true` from `bool` confirm this process.

Even if the example doesn't show it, you can certainly apply the manipulator on input streams like `std::cin` as well—for example:

```

bool b;
std::cin >> std::boolalpha >> b;

```

Here you can now actually enter `true` or `false`, and it will be recognized as a Boolean value.

Manipulators for integers

Other manipulators are available that allow you to set the number base format between decimal, hexadecimal, and octal. In [Table 27.3](#), you will find an overview from the `<ios>` header file.

Manipulator	Reading/writing integral values is done in ...
dec	... decimal number format (default setting).
hex	... hexadecimal number format.
oct	... octal number format.

Table 27.3 Manipulators for the format of integral types.

The following listing shows the three manipulators for changing the integer format.

```
// https://godbolt.org/z/bEKaYdssE
#include <iostream>
#include <ios>
using std::cout; using std::endl;
void f() {
    int val = 100;
    cout << val << endl;           // Output: 0x64
}
int main() {
    int val = 255;
    cout << std::showbase;
    cout << std::dec << val << endl; // Output: 255
    cout << std::hex << val << endl;  // Output: 0xff
    f();
    cout << std::oct << val << endl;  // Output: 0377
    cout << val << std::endl;         // Output: 0377
}
```

Listing 27.6 Number formats in output.

First, I use the `std::showbase` manipulator so that the number base `0x` for hexadecimal or `0` for octal values appears. I could have skipped the manipulator `dec` at this point because it is the default setting, and thus the value is output in decimal form anyway. Next is the `hex` manipulator, which causes the value to be output in hexadecimal notation. The setting then remains global for the stream, as confirmed by the output of the `f()` function. Then the value is output in octal representation after applying the `oct` manipulator. The very last output shows again that the setting remains for the stream. If you want to have the decimal notation again, you must reset the format with the `dec` manipulator.

Manipulators for Floating-Point Numbers

There are also some manipulators (`floatfield` flags) for floating-point numbers, which are also included in `<ios>` and listed in [Table 27.4](#).

Manipulator	Meaning
fixed	Output in floating-point notation: 3.14159.
scientific	Output in exponential notation: 3.14159e+000.
hexfloat	Writes in hexadecimal format: 0x1.921f901b866ep+1.
defaultfloat	Writes the value in the default notation.

Table 27.4 Manipulators for floating-point numbers.

Manipulators for Alignment

Finally, there are various manipulators that allow the stream output to be aligned. An overview of these `adjustfield` flags, which are also defined in `<ios>`, can be found in [Table 27.5](#).

Manipulator	Meaning
<code>internal</code>	Fills the space between the sign and the value.
<code>left</code>	Aligns the output to the left.
<code>right</code>	Aligns the output to the right.

Table 27.5 Manipulators for aligning output.

Using these manipulators to align the output is, of course, only useful if the output has been set to a specific width using `setw()` from the `<iomanip>` header file. For better understanding, here is an example with the corresponding output on the screen:

```
// https://godbolt.org/z/TMdzrjvhd
#include <iostream>
#include <ios>           // left, right, internal
#include <iomanip>       // setw
using std::cout; using std::endl;

int main() {
    int val = -1000;
    cout << std::setw(10) << std::internal
         << val << endl;

    cout << std::setw(10) << std::left << val << endl;

    cout << std::setw(10) << std::right
         << val << endl;
}
```

Listing 27.7 Different ways to pad output.

The program speaks for itself when executed:

```
-      1000
-1000
-1000
```

Manipulators with Arguments

There are some manipulators with arguments in the `<iomanip>` header file. You can find an overview of them in [Table 27.6](#).

Manipulator	Meaning
<code>setiosflags(ios::fmtflags m)</code>	This allows you to combine multiple flags with a bitwise <code> </code> and set them to <code>m</code> .
<code>resetiosflags(ios::fmtflags m)</code>	Clears all flags specified by <code>m</code> .
<code>setbase(int base)</code>	Sets the base for reading and writing integral values to 8, 10, or 16.
<code>setfill(char ch)</code>	If a fill width is used, the empty space is filled with the character <code>ch</code> .
<code>setprecision(int n)</code>	This allows you to set the number of decimal places for floating-point numbers.
<code>setw(int w)</code>	This allows you to set a fill width between two values in a stream.

Table 27.6 Manipulators with parameters from the “`<iomanip>`” header file.

```
// https://godbolt.org/z/sYK8oT33x
#include <iostream>
#include <ios>           // left, right, internal
#include <iomanip>       // setw

using std::cout; using std::endl;
int main() {
    double dval = 3.14159;
    std::ios_base::fmtflags ff(std::ios::scientific|std::ios::uppercase);
    cout << std::setiosflags(ff);
    cout << dval << endl;                // Output: 3.141590E+00
    cout << std::resetiosflags(ff) << dval << endl; // Output: 3.14159
    cout << std::setprecision(3) << dval << endl;  // Output: 3.14
    cout << std::setw(10);
    cout << std::setfill( '*' ) << 1246 << endl;  // Output: *****1246
}
```

Listing 27.8 Number formats in output.

First, I store a few format flags in the bitmask of type `fmtflags`. You can combine multiple flags using the `|` operator. Then I set the flags so that a floating-point number appears in exponent format and the `e` is capitalized. You don't necessarily have to create `ios_base::fmtflags`; you can also write the flags directly into `setiosflags`. Then I

reset these settings with `resetiosflags`. `setprecision()` ensures that the floating-point number is truncated to three digits in the output. The `dval` value is not changed in the process. `setfill()` together with `set()` for the fill width demonstrates how it is extended to 10 characters, and the empty space is now filled with the `*` character.

setw() Is Different

Note that the `setw()` manipulator does not make the width setting permanent, as is the case with all other manipulators. The width is reset after each use and must therefore be set again before each use.

With `get_money()`, `put_money()`, `get_time()`, and `put_time()`, you have four more manipulators. You can use them for locale-specific tasks such as input/output of currency or date and time.

27.5.2 Creating Custom Manipulators without Arguments

You can also create your own manipulators. In practice, you only need to accept the stream as an argument and return a reference to this stream. In between, you can *manipulate* the stream (hence the name *manipulator*). Custom manipulators without arguments can be implemented with a few lines of code, as in the next listing.

```
// https://godbolt.org/z/z8zc3ocd5
#include <iostream>
#include <ios>           // left, right, internal
#include <iomanip>       // setw
using std::cout; using std::cin; using std::endl;
std::ostream& tabl(std::ostream& os) {
    os << '\t';
    return os;
}
std::istream& firstNum(std::istream& is) {
    char ch;
    is.get(ch);
    if( (ch >= '0') && (ch <= '9') ) {
        std::cin.putback(ch);
    }
    return is;
}
int main() {
    int val=0;
    cout << "Text1" << tabl << "Text2" << endl; // Output: Text1(tab) Text2
    cout << "Make an input: ";
```



```
    cin >> firstNum >> val;
    cout << val << std::endl; // Output: 12345
}
```

Listing 27.9 Custom manipulators.

The `tabl` function manipulates `ostream`. You can embed the function in an `ostream` or, more precisely, give it to `operator<<` because an `ostream` is passed as an argument and the `ostream` is also returned by reference. Within the function, you can now modify the stream as desired. For example, you can use one of the many manipulators offered by the standard library. In the example, I wrote a simple `tabl` manipulator that generates a tab character `\t`. You see this in `main` between `Text1` and `Text2`.

You can also manipulate the input, as `firstNum` demonstrates. There I pass an `istream` as an argument and return this `istream` as a reference. Within the function, you can now manipulate the input stream. In this example, I do something similar to what `std::cin` does when reading a leading whitespace—namely, ignore it. In this example, I check the first character of `istream` to see if it matches any of the characters 0 through 9. If that is the case, I push it back into the input stream with `putback`. If it is not one of the characters 0 through 9, an incorrect first character was entered, and I simply let it be “swallowed.” In `main`, you also see an example of this.

The fact that you can pass a custom-written manipulator through the input/output operators is due to the (complex) implementation of the operators:

```
ostream& ostream::operator<<(ostream& (*fp) (ostream& ) ) {
    *fp(*this);
    return *this;
}
```

Thanks to the function pointer `*fp`, our custom-written manipulators are called in the stream. `*fp(...)` then calls the passed function. `fp` is thus a pointer to a function that, as shown in the example, takes an `ostream&` as an argument and returns an `ostream&`. Thanks to automatic conversion of functors and lambdas to C function pointers, it also works as in the following listing.

```
#include <iostream>
using std::cout; using std::endl;
int main() {
    auto ddash = [](auto &os) -> std::ostream& { return os << "--"; };
    cout << "Text1" << ddash << "Text2" << endl; // Output: Text1--Text2
}
```

Listing 27.10 Manipulator as lambda.

Here, instead of a function, I have defined a lambda as a manipulator. Note that you must specify the return type of the lambda with `-> std::ostream&` to ensure that it returns by reference and not by value.

It is the same for the input stream.

27.5.3 Creating Custom Manipulators with Arguments

You can also create manipulators with arguments without much effort using a functor. The following listing is a simple example of how you can write manipulators with parameters.

```
// https://godbolt.org/z/KKE6nWT49
#include <iostream>
using std::cout; using std::endl;
class dendl { // Dots followed by newline
    int dendl_;
public:
    dendl(int n=1)
        : dendl_{n} {}
    std::ostream& operator()(std::ostream& os) const { // Functor
        for(int i=0; i<dendl_; ++i) os << '.';
        return os << '\n';
    }
};
std::ostream& operator<< ( std::ostream& os, const dendl& elem) {
    return elem(os);
}
int main() {
    cout << "Text1" << dendl(4); // Output: Text1....
    cout << "Text2" << dendl(2); // Output: Text2..
    cout << "Text3" << dendl(); // Output: Text3.
}
```

Listing 27.11 Manipulators with parameters.

With `dendl`, you see the typical class definition for defining the functor needed in `main`. This functor itself does nothing more than output `n` dots followed by a newline.

As you can see in the `main` application, you will notice that using the functor alone is not yet possible. The equivalent to `cout << dendl(4)` as a function call is the following:

```
operator<<(std::cout, dendl(4));
```

However, `operator<<` does not yet have an overload for the second parameter of type `dendl`. This must still be provided—as in the example. In it, only the call to the passed function object needs to be made.

The implementation of a manipulator with arguments for the input stream can be implemented similarly.

Reference as Return Value

Here too, it is advisable to return a reference because this ensures that further operations can be chained with the operator.

27.5.4 Directly Change Format Flags

In addition to formatting streams using manipulators, you can also directly change the flags using methods. Although the use of manipulators is generally more common and considerably more convenient, we will briefly discuss the available methods here. The difference is also quite quickly explained. Consider the following listing, in which we use manipulators to format the output.

```
// https://godbolt.org/z/csbPMYM19
#include <iostream>
#include <ios> // hex, dec
using std::cout; using std::endl;
int main() {
    int val = 255;
    cout << std::showbase << std::hex << val << endl; // Output: 0xff
    cout << std::noshowbase << std::dec << val << endl; // Output: 255
}
```

Listing 27.12 Directly influencing format.

First, I use the two manipulators to output the hexadecimal value (`ff`) along with the number base (`0x`). Then I deactivate the output of the number base and switch the base back to decimal.

The manipulators used here are equivalent to the following instructions, where the flags were directly changed using methods.

```
// https://godbolt.org/z/1xzzMdjdM
#include <iostream>
using std::cout; using std::endl;
int main() {
    int val = 255;
    cout.setf(std::ios_base::hex, std::ios_base::basefield);
```

```

cout.setf(std::ios_base::showbase);
cout << val << std::endl; // Output: 0xff
cout.unsetf(std::ios_base::showbase);
cout.setf(std::ios_base::dec, std::ios_base::basefield);
cout << val << std::endl; // Output: 255
}

```

Listing 27.13 Influencing format with “setf” and “unsetf”.

Here the same thing happens as in [Listing 27.12](#), but first the bit for the hexadecimal number representation is set with `hex` in the group `basefield`. To ensure that the number base (0x) is also output, I set this flag for the `cout` stream. I do the same in reverse order further down, where I clear the `showbase` bit again. At the end, I reset the flag in the `basefield` group back to decimal integer output (`dec`).

I think the example shows the clearer and more convenient use of manipulators as opposed to the methods `setf()` and `unsetf()` used here. With manipulators, you don't have to deal with cumbersome things like flags, bits, and bitmasks. And ultimately, a manipulator like `std::showbase` internally calls the `cout.setf(std::ios::showbase)` method anyway.

Some format of the flags can also be used in groups, as seen in the example of `dec` with `basefield`. Specifically, there are three groups with `adjustfield` (and the flags `left`, `right`, `internal`), `basefield` (with the flags `dec`, `oct`, `hex`), and `floatfield` (with the flags `scientific` and `fixed`).

Independent flags, such as `showbase`, can be used alone (though with `ios_base::` or `ios::`). [Table 27.7](#) provides an overview of the methods in the `ios_base` class, which allow you to directly change the format flags.

Method	Meaning
<code>setf(flags)</code>	Sets the format flags to <code>flags</code> .
<code>setf(flags, group)</code>	Sets the format flags of the group <code>group</code> to <code>flags</code> .
<code>unsetf(flags)</code>	Clears all flags.
<code>flags()</code>	Returns the currently set format flags.
<code>flags(flags)</code>	Sets the format flags to <code>flags</code> .

Table 27.7 Methods for accessing format flags.

The next listing is the same example as the previous one, but now everything should be implemented using only the `flags()` method.

```
// https://godbolt.org/z/4MY1Yznv9
#include <iostream>
using std::cout; using std::endl;
int main() {
    int val = 255;
    std::ios::fmtflags ff = std::cout.flags();
    cout.flags(std::ios::hex | std::ios::showbase);
    cout << val << endl; //Output: 0xff
    cout.flags(ff);
    cout << val << endl; //Output: 255
}
```

Listing 27.14 Saving and restoring flags.

I retrieve the current state of the format flags with `flags()` and store it in `ff`. Calling `flags(...)` with the `hex` and `showbase` parameters for the `cout` stream changes the format. Afterward, I restore the previously saved state of the format flags, which the output at the end confirms.

Further Methods for Formatting

You will also find suitable methods in the `ios_base` class (see [Table 27.8](#)) for some manipulators with parameters, primarily to format variable output (like precision or fill width).

Method	Description
<code>int fill() const</code>	Returns the currently set fill character.
<code>int fill(int ch)</code>	Sets the fill character to <code>ch</code> . Equivalent to <code><< setfill(ch)</code> .
<code>int width() const</code>	Returns the field width.
<code>int width(int n)</code>	Sets the field width to <code>n</code> . Equivalent to <code><< setw(n)</code> .
<code>int precision() const</code>	The current precision of floating-point numbers.
<code>int precision(int n)</code>	Sets the precision to <code>n</code> . Equivalent to <code><< setprecision(n)</code> .

Table 27.8 Additional methods of the “`ios_base`” class for formatting.

width() Is Different

As with the manipulator `setw()`, the fill width setting using `width()` does not remain permanent, unlike all other flags. The fill width is reset after each use and must therefore be set again before each use.

27.6 Streams for File Input and Output

You've been using the computer for a while now, and you've found yourself working with variables in the system's memory. You've probably noticed that when you close the program, all those variables just disappear. Well, don't worry, because in this section, I am going to show you how you can save those variables permanently on an external storage device so that you can access them whenever you need them.

Before we get started, I just want to quickly go over how you can access files. To access files, you'll need certain functions. These are provided by the C++ standard library. When reading and writing files, you don't access them byte by byte. Instead, you access them in larger blocks to improve efficiency. For this kind of data transfer, we use a buffer as a temporary storage space in the RAM. You can think of the file or the file's contents as an array of bytes with a specific length.

And just like you write the data into a file, you can also read it back out again. So, you'll need to take care of the formatting and structure yourself. To make sure that reading and writing files works smoothly, there's a special place in the file (starting from 0) that you can use. This is called the *file position*, and it makes sure that files are processed in the right order. If you read 10 bytes from a file, the read position will be on the 10th byte. The same goes for writing.

27.6.1 The “ifstream”, “ofstream”, and “fstream” Streams

The streams for file input and output have been seamlessly integrated into the stream concept of the standard library. If you look at the beginning of the chapter and at the simplified class hierarchy in [Figure 27.1](#), you can see that the `ifstream`, `ofstream`, and `fstream` file streams all have well-known classes as base classes. This means that you can use all the well-known input/output operations (such as the operators `<<` and `>>`) with files as well. Basically, only the stream object changes. Instead of `cout` or `cin`, other stream objects associated with a file are now used. Here is a brief overview of the three available file stream classes:

- `ifstream` is derived from `istream` and is used for reading files.
- `ofstream` is derived from `ostream` and is used for writing to files.
- `fstream` is derived from `iostream` and can be used for reading and writing to files.

For all file streams, you must include the `<fstream>` header file in your program.

27.6.2 Connecting to a File

To do anything with a file, you must first instantiate an appropriate stream object of type `ifstream`, `ofstream`, or `fstream`. Furthermore, you must open the file. You can open the file either directly during instantiation with the appropriate constructor or with the

`open()` method by specifying the file name (optionally with path). You can specify the file name as a C string or `std::string`. Specifying a mode in which you want to open the file is also optional.

No Path Specified

If you do not specify a path, the system assumes that the file is in the same directory where you are running the program.

The next listing shows some examples where a few file streams are created and files are opened.

```
// https://godbolt.org/z/o7YP7or8s
#include <fstream>
#include <iostream>
#include <string>
int main() {
    std::string name = "textfile.txt";
    std::ifstream file01;
    file01.open(name);
    if( file01.fail() ) {
        std::cout << "Could not open " << name << "\n";
    }
    std::ofstream file02("data.dat");
    if( file02.good() ) {
        std::cout << "data.dat opened or created\n";
    }
    std::fstream file03;
    file03.open("database.db");
    if( !file03 ) {
        std::cout << "Could not open database.db\n";
    }
}
```

Listing 27.15 Opening and creating files.

With `ifstream file01`, I create an `ifstream` object for reading using the default constructor. The `file01` stream object created this way is not yet associated with any file. I accomplish this with the `open()` method and the file name. I check whether the file could be opened using the `fail()` method.

Next, I create a new stream object for writing. I provide the file name directly in the constructor. If this file does not exist, it will be created. If this file does exist, it will be truncated to length 0 by default, and the old content will be overwritten. With `good()`, I check whether the file was successfully opened or created.

In the last example, I open a file `database.db` for reading and writing using `fstream`. If the file does not exist, the `!file03` check will return `true` and present an appropriate error message.

Error Checking

For error checking when opening a file, you can use both `if(!file)` and `if(file.fail())`. In both cases, `true` will be returned if opening a file has failed and thus the `failbit` flag is set (see also [Section 27.4](#)). If a file is successfully opened, all error flags are cleared, and the `goodbit` flag is set. Therefore, you can use the `good()` method to test whether the file was successfully opened (`true`) or not (`false`).

Flags for Opening a File

When opening a file stream, you can either stick with the default values of the `fstream`, `ofstream`, or `ifstream` class, or you can set optional flags that determine how the file is opened. In the `<ios>` header file, you will find the definition of the type (specifically the bitmask) `ios::openmode`. The individual flags (see [Table 27.9](#)) can be combined with each other using the bitwise OR operator `|`, where appropriate.

Flag	Short	Description
<code>ios::in</code>	Input	Open for reading (default for <code>ifstream</code>).
<code>ios::out</code>	Output	Open for writing (default for <code>ofstream</code>).
<code>ios::app</code>	Append	Always add something at the end when writing.
<code>ios::trunc</code>	Truncate	The file length is set to 0, and existing content is deleted.
<code>ios::ate</code>	At end	Positions the read or write position at the end of the file immediately after opening. Without this flag, the read and/or write position is usually at the beginning of the file after opening.
<code>ios::binary</code>	Binary	Opens the file in raw binary mode. For example, line-end conversions are not performed, and raw bytes are transferred. Without this mode, a file is opened in a formatted text mode.

Table 27.9 Various flags for opening a file.

Default Values of “ifstream”, “ofstream”, and “fstream”

The default value of `ifstream` is essentially `ios::in`, that of `ofstream` is `ios::out | ios::trunc`, and that of `fstream` is `ios::in | ios::out`.

You can write the flags either directly in the constructor after the file name or in the method after the file name. The following listing shows two simple examples.


```
// https://godbolt.org/z/GoPddTWq6
#include <fstream>
#include <iostream>
using std::cout;
int main() {
    std::ofstream file01("testfile.txt", std::ios::out|std::ios::app);
    if(file01.fail()) {
        cout << "Could not open testfile.txt\n";
    } else {
        cout << "ok.\n";
    }
    std::fstream file02;
    file02.open("database.db", std::ios::out|std::ios::trunc);
    if( !file02 ) {
        cout << "Could not open database.db\n";
    } else {
        cout << "ok.\n";
    }
}
```

Listing 27.16 Additional flags when opening files.

Flags from ios	Meaning
in	Read; file must exist.
out	Truncate and write; file will be created if necessary
out trunc	Truncate and write; file will be created if necessary
out app	Append data; file will be created if necessary
app	Append data; file will be created if necessary
in out	Read and write; file must exist; file position at the beginning
in out trunc	Truncate, read, and write; file will be created if necessary
in out app	Read and write; append data; file will be created if necessary

Table 27.10 Meanings of different combinations for opening a file.

First, I open a file for writing (ios::out) with "testfile.txt" and append the new content to the end of the file (ios::app). I also open the file "database.db" for writing (ios::out), where the content of the file is deleted (ios::trunc). If this file does not yet exist, it will be created.

Confusion often arises about when and how to combine which flags. It can definitely be said that you should always use at least the `ios::in` or `ios::out` flag, which is also the default setting of the classes. For `ios::in`, the file must already exist. If you do not use `ios::in`, the file will be newly created if it does not exist. [Table 27.10](#) provides an overview of various combinations of flags and what they do.

Close File

When you are done with the file, you should return the resources to the system and close the file. First, you do not have an infinite number of file streams available, and second, you ensure that data still in the buffer is finally processed. To close a file, the `close()` method is used. The following listing briefly demonstrates `close()`.

```
// https://godbolt.org/z/TMzdWhE51
#include <fstream>
#include <iostream>
int main() {
    std::ofstream file01("data.db");
    if( file01.fail() ) {
        std::cout << "Could not open data.db\n";
    } else {
        std::cout << "ok.\n";
    }
    file01 << "Text for the file\n";
    if( file01.is_open() ) {
        file01.close();
    }
    file01.open("data001.db");
    // Automatically:
    {
        std::ofstream file02("data002.db");
    } // here file02 is closed
} // here file01 is also closed
```

Listing 27.17 Explicitly closing a file stream.

With `close()`, you cut the connection to the file. To avoid accidentally closing a file that is not (or is no longer) open, I use the `is_open()` method here, which checks whether the stream object `file01` is actually connected to a file. Otherwise, you might trigger an exception when attempting to close.

A file is also closed when the running program terminates or the scope of the stream variable is exited, as you can see with `file01` and `file02`.

You will find an overview of the basic methods for opening and closing files in [Table 27.11](#).

Method	Description
open(filename)	Opens the file for a stream object in the default mode. The default mode depends on the class used (ifstream, ofstream, or fstream).
open(filename, flags)	Opens the file for a stream object and uses flags for the mode.
close()	Closes the file.
is_open()	Returns true if a stream object is open, otherwise false.

Table 27.11 Methods for opening and closing files.

You Cannot Copy Streams by Value

You can't copy stream objects, only move them. This means that you use references as parameters, for example. If you return the parameter, you can also do so by reference. When you create a stream in a function, one possible approach is to return it by value (not by reference). In this case, you must either return a temp-value directly with return or pass a local variable to the outside using `std::move`.

27.6.3 Reading and Writing

After you have opened a file, you can use this stream object for reading and writing, just as you are already accustomed to with standard stream objects like `cin` or `cout`.

The simplest way to write something to a file or read something from it is likely to use the `<<` operator for writing and the `>>` operator for reading. With this, you can also use all known manipulators and methods for formatting. The following example is intended to demonstrate this.

```
// https://godbolt.org/z/3756v8rzK
#include <fstream>
#include <iomanip> // setw
#include <iostream>
int main() {
    std::ofstream file("data.dat");
    if( !file ) {
        std::cout << "Could not open data.dat\n";
        return 1;
    }
    file << std::setw(10) << std::setfill( '*' )
        << 1234 << std::endl;
}
```

Listing 27.18 Reading and writing with files.

With `file("data.dat")`, I open a file for writing. If the file already exists, the old content will be deleted. If this file does not exist, it will be created. Then I push the value 1234 with a field width of ten characters, filling the remaining characters with an asterisk, into the stream object `file`, where everything is written to the file. Afterward, the content of the file `data.dat` looks as follows:

```
*****1234
```

In practice, however, formatted writing and reading with the operators `<<` and `>>` is rarely used. It is much more efficient to read from and write to files in whole blocks and in binary mode. I will go into more detail on the individual possibilities (character-wise, line-wise, and block-wise) in the next sections.

Byte-Wise Reading and Writing

If you want/need to read or write a file byte by byte, you can also use the `get()` and `put()` methods here. The usage is the same as with the stream objects `cin` and `cout`, except that you use the corresponding stream object with which you opened the file.

The following listing reads a file byte by byte and outputs each byte on the screen.

```
// https://godbolt.org/z/sjaGaq48v
#include <fstream>
#include <iomanip> // setw
#include <iostream>
int main() {
    std::ifstream file("data.dat");
    if( !file ) {
        std::cout << "Error opening file\n";
        return 1;
    }
    char ch;
    while(file.get(ch) ) {
        std::cout.put(ch);
    }
    if( file.eof() ) {
        file.clear();
    }
    file.close();
}
```

Listing 27.19 Byte-wise reading and writing.

First, I try to open the `data.dat` file. In the `while` loop, I start with byte-wise reading from the file `file` stream. Each character read with `get()` is immediately output to the screen with `put()`. The `while` loop continues until EOF is reached.

It should now be no problem to copy a file byte by byte instead of outputting to the screen. To do this, you simply need to create a file stream for writing, open it, and write the read bytes there (byte by byte) instead of outputting to `std::cout`. I don't want to withhold the example from you, so examine the following listing.

```
#include <fstream>
#include <iostream>
using std::cout;
int main() {
    std::ifstream file("data.dat");
    if( !file ) { /* Error */ cout << "ERR\n"; return 1; }
    std::ofstream filecopy("backup.dat");
    if( !filecopy ) { /* Error */ cout << "ERR\n"; return 1; }
    char ch;
    while(file.get(ch) ) {
        filecopy.put(ch);
    }
}
```

Listing 27.20 Copying a file byte by byte.

Similarly, you can also read data from the keyboard via the stream object `std::cin` instead of using the data stream `file` and write the typed data into the file stream `filecopy`. However, you must trigger EOF yourself using the key combination `[Ctrl]+[D]` or `[Ctrl]+[Z]`.

Slow Copying

It should be obvious that byte-wise copying is not very efficient.

Line-by-Line Reading and Writing

Processing a file line by line naturally requires files that contain line breaks. Therefore, line-by-line reading or writing is more suitable for formatted files. It makes little sense to process a binary file line-by-line. For reading a stream line by line, the `getline()` method is suitable, and for formatted writing to the stream, the operator `<<()`.

Consider the following listing.

```
// https://godbolt.org/z/hPaz3a3sE
#include <fstream>
#include <iostream>
using std::cout;
int main() {
    std::ifstream file("44fstream07.cpp");
    if( !file ) { /* Error */ cout << "ERR\n"; return 1; }
```

```

std::ofstream filecopy("backup.cpp");
if( !filecopy ) { /* Error */ cout << "ERR\n"; return 1; }
std::string buffer;
while( getline(file, buffer) ) {
    filecopy << buffer << std::endl;
    cout << buffer << std::endl;
}
if( file.eof() ) {
    file.clear();
}
}

```

Listing 27.21 Line-by-line reading and writing.

This listing largely corresponds to the previous example, in which data was read byte by byte. Now you will find `std::string` instead of `char` as a buffer, into which I read a line from the file file stream using `getline()` in the while loop. `filecopy << buffer` copies this read line into the `filecopy` file stream. The newline is automatically removed by `getline()`, so you need to add a newline to the stream here. I also output the line to `std::cout` (usually the screen). The while loop reads line by line until EOF is reached.

Block-Wise Reading and Writing

For reading and writing entire raw data blocks, the `read()` and `write()` methods are best suited.

The following listing allows you to copy a file in one go. Random access is also used here, which will be explained in the next section.

```

// https://godbolt.org/z/P6MPabsKc
#include <fstream>
#include <iostream>
#include <vector>
using std::cout;
int main() {
    std::ifstream file("testfile.txt", std::ios::binary);
    if( !file ) { /* Error */ cout << "ERR1\n"; return 1; }
    std::ofstream filecopy("backup.dat", std::ios::binary);
    if( !filecopy ) { /* Error */ return 1; }
    file.seekg(0, std::ios::end);
    auto size = file.tellg();
    cout << "File size: " << size << " bytes\n";
    file.seekg(0, std::ios::beg); // Important!
    std::vector<char> buffer(size);
    file.read(buffer.data(), size);
    if( !file ) { cout << "Error during read...\n"; return 1;}
}

```

```

cout << "Read: " << file.gcount() << " bytes\n";
filecopy.write( buffer.data(), size );
if( !filecopy ) { cout << "Error during write...\n"; return 1;}
}

```

Listing 27.22 Block-wise reading and writing with “read” and “write”.

With `file` and `filecopy`, I first opened a file for reading and another file for writing to copy (both in binary mode with `ios::binary`). Then I set the position of the file to be copied to the end (`ios::end`) using the `seekg()` method. Using the position at the end of the file, I get the byte offset from the beginning of the file with the `tellg()` method and later store the size of the file in the `size` variable for copying.

It is also very important now that I reset the read position to the beginning with `seekg`. Then I get a sufficiently large buffer to read the entire file. For this, I initialize a vector of size `size` with elements of type `char`. A vector guarantees that all elements are stored consecutively in one block. This is useful and necessary to serve as a parameter for the following calls to `read` and `write`. For this very purpose, vector has the `data()` method.

Often at this point, a dynamic C-array is fetched—for example, with `new char[size]`. That works too, but you shouldn't forget the `delete[]` then. `unique_ptr` can offer support here, but vector already has everything needed in one package, so I like to use it for this purpose. Later in [Listing 27.29](#), I will show you the `char[]` alternative.

Storing Vectors, Structures, or Classes

Although the listing only demonstrated how to copy a simple file in blocks, in practice, the ability to use `read()` and `write()` is also well suited for storing more complex data structures such as vectors, structures, or the properties of classes. It is always necessary to ensure that the type must be converted to and from `char*`.

Sooner or later, you will want to store the properties of classes. As this seems somewhat more complex, a simple recipe for practice will be shown here. As a basic framework, I use the class in the following listing.

```

// https://godbolt.org/z/b3W48s86z
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using std::cout; using std::string;
class DataClass {
    std::string text_;
    int data_;
public:

```

```

DataClass(string t="", int i=0)
    : text_{t}, data_{i} {}
std::ostream& write(std::ostream& os) const {
    os << text_ << std::ends;
    os.write(reinterpret_cast<const char*>(&data_), sizeof(data_));
    return os;
}
std::istream& read(std::istream& is) {
    std::getline(is, text_, '\0');
    is.read(reinterpret_cast<char*>(&data_), sizeof(data_));
    return is;
}
std::ostream& print(std::ostream& os) {
    return os << text_ << " : " << data_ << std::endl;
}
};
int main() {
    std::ofstream file_w("data.dat", std::ios::binary);
    if( !file_w) { cout << "Error opening file\n"; return 1; }
    std::vector<DataClass> vec_dat;
    vec_dat.push_back(DataClass("A text", 123));
    vec_dat.push_back(DataClass("More text", 321));
    vec_dat.emplace_back("Much more text", 333);
    for(const auto &elem : vec_dat){
        elem.write(file_w);
    }
    file_w.close();
    std::ifstream file_r("data.dat", std::ios::binary);
    if( !file_r) { cout << "Error opening file\n"; return 1; }
    DataClass dat_r;
    while( file_r ) {
        dat_r.read(file_r);
        if( file_r.eof()) break;
        dat_r.print(cout);
    }
}

```

Listing 27.23 Block-wise reading and writing with a helper class.

The `DataClass` class contains two data fields that should be written and read in binary form. The main focus here is on the implementation of the two `write()` and `read()` methods.

At the method head of `write`, you can see that you are writing the properties of `DataClass` to the `os` stream and returning it. `text_` is simply written away with `<<`, but the

`std::ends (='\0')` manipulator is appended as an end marker, which we need again to read the data.¹ Keep in mind that `text_` is a subobject of the `DataClass` class, whose length can always vary.

`data_` refers to `char` elements as in `string`. The data from `int` must first be converted into a raw data block of `const char*` because `write()` expects this as the first argument. This is done using `reinterpret_cast<const char*>`. `const` is used because the method is `const`. This means `data_` must not be changed. A conversion to `char*` would not work, but `const char*` would—thus ensuring that no one changes the data.²

Similarly, this also works with the counterpart `read()`, just in the other direction. The method takes an input stream as a parameter and returns it. To read the first `text_` property, I use `getline()`. Here I read up to the `'\0'` character. I had previously appended this specifically for `write()`. When reading, you also need to convert the first argument to raw data for `read()` as `read()` expects it. This is done using `reinterpret_cast<char*>` as the data is now changing.

In `main`, I then write away three `DataClass` elements, which I retrieve from a previously filled vector. For demonstration purposes, I read them back immediately and output them on the console—for your and my reassurance.

27.6.4 Random Access

In block reading in the previous section (in [Listing 27.22](#)), you have already become familiar with random access to file streams. This makes it possible to determine the current write or read position or to change the position. In [Table 27.12](#), you will find an overview of the available methods.

Method	Description
<code>istream& seekg(streampos), istream& seekg(streampos, ios::seekdir)</code>	Sets the read position to a specific position from the beginning of the file or relative to the optionally specified second argument
<code>streampos tellg()</code>	Returns the current read position
<code>ostream& seekp(streampos), ostream& seekp(streampos, ios::seekdir)</code>	Sets the write position to a specific position from the beginning of the file or relative to the optionally specified second argument
<code>streampos tellp()</code>	Returns the current write position

Table 27.12 Methods for random access.

1 In practice, `string` can contain null characters. If that is possible, you must proceed differently and, for example, write down the length.
2 If the person does not cheat maliciously.

The possible values for the second argument of `seekg()` and `seekp()` (g stands for get and p for put) for the reference position of type `ios::seekdir` are as follows:

- `ios::beg`: position relative to the beginning of the file
- `ios::cur`: position relative to the current position in the file
- `ios::end`: position relative to the end of the file

Here are some examples:

```
// Position 10 characters forward from the current position
file.seekg(10, std::ios::cur);
// ...
// Position at the beginning of the file
file.seekg(0, std::ios::beg);
// ...
// Position 5 characters before the end
file.seekg(-5, std::ios::end);
```

You must ensure that you set the file position to a valid range here. If you choose a position before the beginning or after the end, then the behavior is undefined.

27.6.5 Synchronized Streams for Threads

When working with multiple threads, outputs of the threads can overlap each other. To prevent a thread from interrupting the output of a stream, you need a synchronization mechanism. You could use a mutex with a `unique_lock` at all output points. Since C++20, there is a simpler solution for this: in the `<syncstream>` header, there is the `osyncstream` class, which works similarly to `unique_lock`—following the RAII pattern. But instead of introducing an additional mutex, you use the respective stream whose output you want to synchronize as a parameter.

```
// https://godbolt.org/z/GefKP5sj4
#include <iostream>
#include <thread>
#include <syncstream>
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
void runFib(long from, long step, long to) {
    for (auto n=from; n<=to; n+=step) {
        std::osyncstream osync{ std::cout }; // Sync on cout as long as osync exists
        osync << "fib("<n<<"=" << fib(n) << '\n';
    }
}
int main() {
    std::jthread f40{ runFib, 1, 3, 40 };
    std::jthread f41{ runFib, 2, 3, 41 };
```

```
std::jthread f42{ runFib, 3, 3, 42 };  
}
```

Listing 27.24 Synchronizing output between threads with “osyncstream”.

Here, the Fibonacci numbers are calculated in three threads in steps of three from 1 to 42. The variable `osync` locks the output to `std::cout` for other threads for as long as it exists—but only if the other threads also use `osyncstream`. Apart from the order, the output is nice and clean line by line:

```
fib(2)=1  
fib(5)=5  
fib(8)=21  
fib(11)=89  
fib(14)=377  
...
```

If you did not synchronize the output, then the outputs would almost certainly be interrupted, especially before `<< fib(n)`, because the calculation takes time. This is how it would look without `osyncstream`:

```
fib(fib(3)=2)=21  
fib(5)=5  
fib(8)=21  
fib(11)=89  
...
```

The `osyncstream` class is on one hand a wrapper for a stream, but on the other hand also a stream itself. You can use it like a stream, so you can output it with `<<` as in the example. But it is also suitable as a parameter for functions that expect an `ostream&`. However, you must pass it as `std::ref(osync)` when calling. If you want to force an output before the destructor of `osync`, output `<< std::flush_emit`.

You can synchronize not only `cout` or other standard streams, but also file streams that you open with `ofstream`.

27.7 Streams for Strings

If you look at the simplified class hierarchy of the stream class in [Figure 27.1](#), starting from `ios`, you will find three streams for strings: `istringstream`, `ostreamstream`, and `stringstream`. For *strings* means that you are reading or writing a string rather than outputting to or from a console or file.

For all three classes, you need to include the `<sstream>` header. These streams are well-suited for converting different data types to and from strings—again, with all the

advantages of the preceding streams and their methods and manipulators. [Table 27.13](#) lists the various string streams.

Stream	Meaning
<code>istringstream</code>	Creates a string stream that can be read from. The stream is opened in <code>ios::in</code> mode.
<code>ostringstream</code>	Creates a string stream for writing. By default, this stream is used with the <code>ios::out</code> mode.
<code>stringstream</code>	Creates a string stream for reading and writing. The default modes here are <code>ios::in</code> and <code>ios::out</code> .

Table 27.13 String streams of the standard library.

For all three constructors, you can use an empty constructor as an argument; the open mode, as with file streams (see [Table 27.9](#)); a string (`std::string`) with which the stream should be initialized immediately; or a combination of the string (first argument) and the open mode (second argument). It should be added to the open modes that for string streams, `ios::in`, `ios::out`, and `ios::ate` make the most sense. The other values of the type `ios::openmode` are also implemented, but whether they actually have an effect on a `stringstream` object depends on the implementation.

Old `strstream`

In the old C++98 standard, a stream class was still implemented with the `char*` type. Today, the `string` type (more precisely, `basic_string`) is used for this purpose. The old string stream classes `istrstream`, `ostrstream`, and `strstream` are therefore only present for backward compatibility reasons and should no longer be used in new projects; they are *deprecated*, meaning that they are obsolete.

Because the string streams inherit all public methods from the base classes derived from them, all previously known and introduced methods with the string streams are also available to you here. However, the main methods for string streams are provided by the class itself with the `str()` method (see [Table 27.14](#)).

Method	Description
<code>string str() const</code>	Returns the string stream buffer as a string
<code>void str(const string& s)</code>	Resets the content of the string stream buffer

Table 27.14 “`str()`” is the centerpiece of the string stream.

This way, you can convert simple data types like `double` or `int` into a string, as in the following listing.

```
// https://godbolt.org/z/TT13E3sz5
#include <sstream> // ostringstream
#include <iostream>
int main() {
    std::ostringstream ostr;
    double dval = 3.1415;
    int ival = 4321;
    ostr << dval << " : " << ival;
    const std::string sval = ostr.str();
    std::cout << sval << std::endl; // Output: 3.1415 : 4321
}
```

Listing 27.25 Writing to a “stringstream”.

First, I create the string stream for writing, which we need for the conversion. Then I simply write the double value and the int value separated by a colon into the string stream using <<. With the str() method, I copy the entire content of the string stream buffer into the sval string and then output it.

The same works in the other direction. The following listing shows how you can extract individual types from a string.

```
// https://godbolt.org/z/1EMox3hW5
#include <sstream> // istringstream
#include <iostream>
int main() {
    std::istringstream istr;
    std::string sval("3.1415 : 4321");
    std::string none;
    double dval=0.0;
    int ival=0;
    istr.str(sval); // initialize
    istr >> dval >> none >> ival; // read
    if( ! istr.eof() ) {
        std::cout << "Error converting\n"; return 1;
    }
    std::cout << dval << " == " << none << " == " << ival << "\n";
    // Output: 3.1415 == : == 4321
}
```

Listing 27.26 Reading from a “stringstream”.

First, I initialize an empty istringstream in istr, from which I will later extract values. I then set the content with str(...). You could also pass the content directly to the constructor.

Extraction is done as with any other `istream` using `>>`. This means that for `double`, characters are consumed so long as they could be part of a floating-point number. Extraction stops at a whitespace. The colon `:` ends up in `none`, because when reading into a string, `>>` also stops at the next whitespace. Finally, it reads into an `int` so long as there are 0 to 9 or `+` and `-` in the string stream.

With `eof`, I check if any errors occurred during the conversion. If the string stream is not at the end, that would be the case.

Conversion from binary to ASCII and vice versa is quite easy with string streams. However, because the conversion work is almost always quite similar, it is advisable to create a function template for this purpose. The following function template serves as a simple recipe that you can use to convert a C++ basic data type to a string or a string to a basic data type.

```
// https://godbolt.org/z/vcoEcYYcx
#include <sstream> // stringstream
#include <iostream>
#include <stdexcept> // invalid_argument
template <class T1, class T2>
void myConvert(const T1& in, T2& out) {
    std::stringstream ss;
    ss << in;
    ss >> out;
    if( ! ss.eof() ) {
        throw std::invalid_argument("Error during conversion");
    }
}

int main() {
    std::string sval;
    float fval=3.1415f;
    std::string sdval("5.321");
    double dval=0;
    std::string doesnotwork("does not work");
    try {
        myConvert(fval, sval);
        std::cout << sval << std::endl; // Output: 3.1415
        myConvert(sdval, dval);
        std::cout << dval << std::endl; // Output: 5.321
        myConvert(doesnotwork, dval); // triggers "Error during conversion"
    }
    catch(const std::invalid_argument& e) {
        std::cout << e.what() << std::endl;
    }
}
```

Listing 27.27 Type conversion using “stringstream”.

The main focus in this example is on the `myConvert` function template. This should convert values of type `T1` to type `T2`. For this, I simply send the data with `<<` into the string stream `ss` and then retrieve it with `>>`.

To check if everything worked, I use `!eof()` and throw an `invalid_argument` exception in the case of an error.

In the `try` block in `main`, a `float` value is converted to a string and then a string value to a `double`. The third conversion throws an exception because converting the text sequence "does no work" to a `double` fails.

String Streams Instead of File Streams

String streams can indeed be used excellently for converting to and from strings, but in practice, they are even more frequently used for buffering purposes to avoid temporary files. Basically, you just need to use a corresponding string stream instead of a file stream. The advantage is obvious: string streams are significantly faster (as they are in main memory) than slow temporary files, which require rather slow access to a storage medium. However, this depends on the use case. It should also be clear that data that only resides in the buffer of the main memory will be lost in the event of a program crash.

27.7.1 Difference from “to_string”

With `ostringstream`, you can convert all objects to a string for which you or someone else has defined `operator<<` for `ostream&`.

Alternatively, there is also the `to_string(...)` free function with predefined overloads for all built-in integer and floating-point types. However, the output may differ in detail from what would appear on an `ostream`. The implementation of the `to_string` overloads is based on what the `printf` C function would output and uses its `%` formats.

```
// https://godbolt.org/z/z6sYqM48e
#include <iostream>
#include <string>
void show(double f) {
    std::cout << "os: " << f
               << "\t to_string: " << std::to_string(f) << "\n";
}
int main() {
    show(23.43);      // Output: os: 23.43  to_string: 23.430000
    show(1e-9);       // Output: os: 1e-09  to_string: 0.000000
    show(1e40);       // Output: os: 1e+40  to_string: 100...0752.000000
    show(1e-40);      // Output: os: 1e-40  to_string: 0.000000
```

```
    show(123456789); // Output: os: 1.23457e+08 to_string: 123456789.000000
}
```

Listing 27.28 The “to_string” function.

The output of `1e+40` with `to_string` has been abbreviated here with

27.7.2 “to_chars” and “format” Are More Flexible than “to_string”

Because you cannot influence the detailed output format with `to_string`, it is of little use, especially for floating-point numbers.

Since C++17, there is the `to_chars` function (*to-char-sequence*), which allows you to write a number into a preinitialized string. This function has additional overloads for further formatting options.

The most versatile way to convert to a string is via the `format` function, which has been available since C++20.

27.7.3 Reading from a String

If you want to convert a string to a number, you can again use `istringstream` and read from it with `>>`. There are overloads of `operator>>(istream& is, ...)` for all built-in numeric types.

Alternatively, there is a whole family of functions for directly converting a character string to a number. You can see a list in [Table 27.15](#).

Function	Numeric Type
<code>stoul</code>	unsigned long
<code>stoull</code>	unsigned long long
<code>stoi</code>	int
<code>stol</code>	long
<code>stoll</code>	long long
<code>stof</code>	float
<code>stod</code>	double
<code>stold</code>	long double

Table 27.15 Conversion functions for strings to numbers.

27.8 Stream Buffers

Although you will rarely need to access them directly, the stream buffer classes should be briefly described here to conclude the stream classes. If you look at the class hierarchy in [Figure 27.1](#), you will find them somewhat pushed to the edge with `streambuf`, `filebuf`, and `stringbuf`. In reality, however, these stream buffer classes are more like the core of the I/O stream library. If you have always wondered where the individual bytes of the stream classes are processed, you will find the answer in the stream buffer classes.

The stream buffer classes provide basic methods for buffering streams without extensive formatting options. Every stream (string stream, file stream, and standard I/O stream) works with stream buffers. The abstract `streambuf` base class is used by all other parts of the I/O stream classes. For buffering the input and output of files, `filebuf` is used, and for buffering strings in memory, `stringbuf` is used. Both are derived from `streambuf`.

With buffering, individual characters are not sent to the stream; instead, the data is transmitted in blocks. The output occurs only when a block unit of the buffer is full. However, you can also manually intervene to explicitly empty the buffer using the `flush()` method, for example:

```
#include <fstream>
int main() {
    std::ofstream file("logfile.log");
    file << "Text for the log";
    file.flush();
}
```

If the data has not yet been transferred to the file stream `file` because the block unit of the buffer was not yet ready, a simple call to the `flush()` method will handle it.

However, `flush()` is implicitly called anyway when a stream is closed. And that, in turn, happens automatically in the destructor—that is, when the stream is removed, here when exiting `main`.

Application Area

For normal household use, you will rarely access the stream buffer classes directly. They play a more important role when you want to create new classes of streams for input and output (e.g., sockets or GUIs). Or if you want to redirect streams, direct access to the stream buffer is very useful. Likewise, it is possible to combine streams using stream buffer classes (e.g., encrypting the output before it is written to another stream).

27.8.1 Access to the Stream Buffer of “iostream” Objects

The most important thing for you to know is that for every `iostream` object, there is a pointer to a `streambuf` object. Using this pointer, it is always possible to directly access the raw bytes in the stream buffer. There are several methods that can be called with a `streambuf` object.

To access a stream buffer, each I/O stream object provides a method called `rdbuf()`, which returns a pointer to the `streambuf` object. With this, you can now access the `streambuf` methods. For example, you can connect the `streambuf` pointer to another stream by using `operator<<`:

```
std::ifstream file("logfile.log");
std::cout << file.rdbuf();
```

With `cout << file.rdbuf()`, the entire content of the file is pushed to `cout` and displayed in one go. This method of passing the entire content to another stream in one go is even significantly more efficient than outputting the entire content using `read()` and `write()`, as you did in [Listing 27.23](#).

In the following listing, I will show you how to place the contents of a file from the buffer into a raw `char` array and possibly further process it there. In the example, we simply output the content to the screen with `write()`.

```
// https://godbolt.org/z/8hK9Y4vhd
#include <fstream>
#include <iostream>
#include <memory> // unique_ptr

int main() {
    std::fstream file("27Streams.tex");           // Open file for reading
    auto bufptr = file.rdbuf();                  // std::streambuf*
    auto size = bufptr->pubseekoff(0, file.end);   // std::streamsize
    bufptr->pubseekoff(0, file.beg);              // back to the beginning
    auto buffer = std::unique_ptr<char[]>(new char[size]); // allocate memory
    auto n = bufptr->sgetn(buffer.get(), size);    // transfer number of chars
    std::cout << "Characters read: " << n << "\n";
    std::cout.write(buffer.get(), size);         // Output char[]
}
```

Listing 27.29 Transferring data from the “`rdbuf`”.

Thanks to `bufptr = file.rdbuf()`, you now have full control over the `streambuf` buffer with the `streambuf` pointer. First, I use `pubseekoff` and `file.end` to set the stream buffer pointer to the end of the buffer to obtain the size buffer size. Because I also want to read from the buffer afterward, I then set the position of the stream buffer pointer back to the beginning of the buffer with `file.beg`.

In a `unique_ptr<char[]>`, I now reserve memory from the heap in which I can store the entire contents of the buffer. With the `sgetn()` method, I now copy `size` bytes from the stream buffer into the `char` array. That's it. In this way, I would have transferred the entire stream buffer into a raw `char` array. For demonstration purposes, I use `cout.write(...)` to output the contents of `char[]` to the console.

This time I used `char[]` as a data buffer together with a `unique_ptr`. Without `unique_ptr`, you would have had to take care of `delete[]` in a larger program. This solution here is an alternative to using a vector as a buffer, as you saw in [Listing 27.22](#).

You now know how to use the `rdbuf()` method to get a stream buffer (of any kind). With the returned pointer, you can now use various methods within the stream to change the position, read characters or entire blocks, and write or reset characters in the stream buffer.

27.8.2 “filebuf”

The `filebuf` class derived from `streambuf` also contains methods to directly open a file and create a stream buffer. The following code snippet is intended to briefly demonstrate this:

```
std::ofstream file;
std::filebuf* bufptr = file.rdbuf();
bufptr->open("file.txt", std::ios::out|std::ios::app);
const char text[]="This goes into the file";
bufptr->sputn(text, sizeof(text)-1);
bufptr->close();
```

In this example, I open or create a file named `file.txt` and directly associate the file content with the file buffer to perform direct operations on it. In the example, I use the `streambuf` method `sputn()` to write the text `text` into the buffer and simultaneously into the file.

27.8.3 “stringbuf”

`stringbuf` contains all the methods from the `streambuf` base class. It also contains the `str()` method with the same meaning as in the class `stringstream`, but here in connection with a string stream buffer.

27.9 “filesystem”

The `<filesystem>` header deals less with the contents of files and more with files as a whole and their relationship to each other. Since C++17, it has been part of the standard; previously, it was extensively tested in `Boost.Filesystem`.

All components of the header are located in the `std::filesystem` namespace. The most important components are as follows:

- **Class `path`**

Represents the name of a file or directory. A path can be easily converted into a string or, starting from C++20, into a `u8string`. With `append`, you add a path component; with `concat`, you modify the last path component.

- **Class `directory_entry`**

Contains metadata about a file or directory.

- **Classes `directory_iterator` and `recursive_directory_iterator`**

Allow iterating over all elements of a directory.

- **Helper functions**

Many free functions to facilitate working with permissions, creating directories, copying or deleting entire files, and so on.

In [Listing 27.30](#), you can see an example of how to concatenate paths and obtain some basic information about files.

```
// https://godbolt.org/z/jddKxj7fW
#include <filesystem> // std::filesystem
#include <iostream>
namespace fs = std::filesystem; using std::cout; using std::endl;
int main() {
    // Path components
    fs::path root {"/"};
    fs::path dir {"var/www/"};
    fs::path index {"index.html"};
    // concatenate
    fs::path p = root / dir / index; // operator/
    // output
    cout << "Name: " << p << endl; // "/var/www/index.html"
    // decompose
    cout << "Parent: " << p.parent_path() << endl; // "/var/www"
    cout << "Name: " << p.filename() << endl; // "index.html"
    cout << "Extension: " << p.extension() << endl; // ".html"
    // Information
    cout << std::boolalpha;
    cout << "Exists? " << fs::exists(p) << endl;
    cout << "Regular file? " << fs::is_regular_file(p) << endl;
}
```

Listing 27.30 How to concatenate paths.

Using operator/ to concatenate paths is clever and looks good. With `p /= ...`, you could directly append an element to `p`. Both correspond to `append`.

When you use `concat`, `+`, or `+=`, you extend the last path component, adding without `"/"`.

Windows, Unix, and Embedded

On some extremely lightweight platforms, `<filesystem>` will not be available to you. However, the authors of this part of the standard library have managed to unify both the Windows and Unix worlds.

This part of the library is also somewhat heavier and likely not implemented as “header-only.” I read a note that the `-lstdc++fs` option when linking with GCC and `-lc++fs` with Clang must be specified. However, this was no longer the case for me.

To give you an idea of the capabilities of `filesystem`, I will provide an excerpt from the long list of features:

- **copy_file, rename, remove, remove_all**
Simple copying or renaming of a file as well as removing a single file or recursively all files in a directory.
- **current_path**
Returns the current directory.
- **exists, equivalent**
`exists` checks if a file exists; `equivalent` checks if two different path instances refer to the same file.
- **file_size, is_empty, is_regular_file**
Provide various information about a file.

27.10 Formatting

Since C++20, there is a new way to format data. In the `<format>` header, the following elements are the most important:

- **format**
Formats any number of arguments into a `string` according to a provided format.
- **format_to**
Formats into a buffer or iterator.
- **format_to_n**
Formats into a buffer up to a specified length.
- **format_size**
Determines the length of a formatted output so you can allocate enough space for a buffer.

- **vformat, vformat_to and make_format_args**
Like `format` and `format_to`, but the format string does not need to be known at compile time. The elements to be formatted must then be wrapped with `make_format_args`.
- **format_error**
The exception that can be thrown during formatting.
- **formattable**
A concept for types that you can format.
- **formatter**
These template classes perform the actual formatting and are your entry point into formatting custom types.

27.10.1 Simple Formatting

The return of the `format` function is a string. The first parameter is always the format of type `format_string`. This must be known at compile time.

I'll Keep It Short

The return value can also be a `wstring`, and the format can be a `wformat_string` as well. And optionally, you can specify a `locale` before the format. I will not go into further detail in this section.

Most of what is said in this section can be applied to `format_to`, `format_to_n`, `vformat`, and `vformat_to`.

A format string contains curly bracket pairs `{}` as placeholders where the arguments should be inserted. The rest of the format string is taken unchanged. An index can be placed inside the brackets. This index refers to the additional arguments to `format` and indicates which argument should be inserted at this position. Optionally, format attributes that affect the representation of the argument can follow the index, separated by a colon `:`.

It depends on the `formatter` class how exactly the attributes are evaluated. The standard formatters are based on Python's format specification and can also be transferred to the classic `printf` syntax:

```
// https://godbolt.org/z/hTj94enjK
#include <format>
#include <chrono>
#include <string>
#include <string_view>
#include <iostream>
using namespace std; using namespace std::literals;
```

```
void pr(string_view s) { cout << s << endl; }
double pi = 3.14159265359;

int main() {
    pr(format("Hello, {}!", "Reader"));           // simple C-string
    pr(format("Hello, {}!", "Author"s));          // simple string
    pr(format("You are {} years old.", 30));        // integers
    pr(format("That makes {:.2f} euros.", 19.9933)); // 2 decimal places
    pr(format("Scientific: {:e}", -44.876));        // results in "-4.487600e+01"
    pr(format("Binary of {} is {:b}.", 42, 42));    // binary without base
    pr(format("Hex of {} is {:#x}.", 73, 73));      // hexadecimal with base
    pr(format("Zero-padded: {:03}", 7));           // results in "007"
    pr(format("|{0:<10}|{1:^10}|{2:>10}|", "le", "mi", "ri"));
    // Alignment and index
    pr(format("{} {:.9}!", "Boa", "Constrictor")); // without index, truncate string
    using namespace std::chrono;                  // neat time specifications:
    pr(format("{} {}", 2023y/11/5, minutes{20})); // Output: 2023-11-05, 20min
}
```

In the last example, you see that you can also format your own types. Here the `<chrono>` header brings in its own formatter.

By the way, did you notice the new C++20 notation for date literals in `2023y/11/5`?

In general, the standard formatting attributes are structured as follows; these are the possible elements in a compact form:

fill align sign # 0 width .precision L type

All elements are optional, but the order must be maintained. The elements are partially dependent on each other and only make sense in certain combinations and for certain types. There are combinations that are frequently seen, such as `.2f` for a floating-point number with two decimal places.

The individual elements mean the following:

- **fill**
Specifies the character to be used for filling. The default is the space character.
Requires *align*.
- **align**
Can be `<` (left), `>` (right), or `^` (centered). The default is left.
- **sign**
 - includes the sign only for negative numbers (default), `+` always includes the sign, and *space* means that a minus sign is used for negative numbers and a space for positive numbers.

- **#**
Enables alternative formats, such as prefixes for integers or points for floating-point numbers.
- **0**
Pads numbers with zeros.
- ***width***
Specifies the minimum width that the result should have. The default is zero.
- ***.precision***
Decimal places for floating-point numbers or how many characters of the string argument should be used.
- **L**
Enables locale-dependent formatting for numbers and Booleans.
- ***type***
Switches the output to a specific type: *d* for integers, *f* for floating-point numbers, and so on.

27.10.2 Formatting Custom Types

To use instances of your own class as an argument for `format`, you need to specialize the class template `std::formatter<>` for your class. This class must then provide two methods with the correct signature:

- **Parse**
Evaluates format attributes and returns an iterator to the first character after the parsed format attributes.
- **Format**
Writes the formatted output of the argument to the output and returns the iterator to the end of the output.

The class also must have a default constructor and must be copyable as well as assignable.

```
// https://godbolt.org/z/sdWsnTonG
#include <format>
#include <string>
#include <vector>
#include <iostream>
struct Elf {
    std::string name;
    int birth_year;
    std::string era;
    std::string folk;
};
```



```
template<> struct std::formatter<Elf> {
    std::formatter<std::string> sub_fmt;
    constexpr auto parse(std::format_parse_context& pctx) {
        return sub_fmt.parse(pctx); // returns iterator to '}'
    }
    auto format(const Elf& elf, std::format_context& fctx) const {
        std::string s = std::format("{} / {} ({} {})",
            elf.name, elf.folk, elf.birth_year, elf.era);
        return sub_fmt.format(s, fctx); // delegate formatting
    }
};

int main() {
    std::vector<Elf> elves{
        {"Feanor", 1169, "First Age", "Noldor"},
        {"Galadriel", 1362, "EZ", "Noldor"},
        {"Legolas", 87, "DZ", "Sindar"},
        {"Elrond", 532, "EZ", "Half-elf"},
        {"Elwe", 1050, "EZ", "Sindar"},
    };
    for (const auto& e : elves) {
        std::cout << std::format("Elf: {:>20}", e) << std::endl;
    }
}
```

Listing 27.31 A formatter can delegate parsing and formatting.

Here I have delegated both the parsing of format attributes and the formatting of the output to the standard formatting for `std::string` to `sub_fmt`. Therefore, I need to somehow generate a string in string `s` that contains the output. I make this easy for myself here by building a string with `std::format` and then passing it to `sub_fmt`.

The output looks like this for Feanor:

```
Elf: Feanor/Nordor (1169 EZ)
```

If you need more sophisticated formatting, you have to parse the format attributes yourself. When parsing, assume that you have an iterator at `pctx.begin()` pointing to the first character after the bracket `{` or after the colon `:`, which marks the beginning of the format attributes. You then need to parse the attributes and return the iterator to the bracket `}`.

In `format`, the output occurs after `fctx.out()`, and you return an iterator to the end of the output.

```

// https://godbolt.org/z/WqjK31Gn6
// 'Elf' as before...
template<> struct std::formatter<Elf> {
    std::string attribs; // Sequence of '%n', '%g', '%e', '%v', and others
    constexpr auto parse(std::format_parse_context& pctx) {
        auto it = std::ranges::find(pctx.begin(), pctx.end(), '}'); // search for '}'
        attribs = std::string(pctx.begin(), it); // save everything
        return it; // points to '}'
    }
    auto format(const Elf& elf, std::format_context& fctx) const {
        auto out = fctx.out(); // into here
        for(auto n=0u; n<attribs.size()-1; ++n) {
            if(attribs[n] == '%') { // instruction to output a member
                switch(attribs[+n]) {
                    case 'n': out = std::format_to(out, "{}", elf.name); break;
                    case 'g': out = std::format_to(out, "{}", elf.birth_year); break;
                    case 'e': out = std::format_to(out, "{}", elf.era); break;
                    case 'v': out = std::format_to(out, "{}", elf.folk); break;
                    case '%': out = std::format_to(out, "%"); break; // %% becomes %
                }
            } else {
                out = std::format_to(out, "{}", attribs[n]); // everything else
            }
        }
        return out; // points to the end
    }
};

int main() {
    Elf e{"Feanor", 1169, "EZ", "Nordor"};
    std::cout << std::format("{}:Elf %n", e) << std::endl;
    // Output: Elf Feanor
    std::cout << std::format("Elf {:%n, %v, born %g in the age %e}\n", e);
    // Output: Elf Feanor, Nordor, born 1169 in the age EZ
}

```

Listing 27.32 A formatter can parse and output itself.

Here I parse the format string up to the `}` and store everything in `attribs`. When outputting in `format`, I iterate over `attribs` and interpret a percent sign `%` as an instruction to output a member of `Elf`. Everything else is simply output.

The idea of using the percent sign as an instruction is not new and is also used, for example, for types from `<chrono>`.

If you want to report an error in parsing or output, throw a `std::format_error` exception.

Chapter 28

Standard Library: Extras

Chapter Telegram

- **pair and tuple**
Anonymous structures that can be easily created on the spot—for example, for returning multiple values.
- **Regular expression**
Formal description of a pattern for text search.
- **Random number generator**
A mechanism that generates random values.
- **Random distribution**
Defines the probabilities via which a random number generator produces values.
- **time_point and duration**
Type-safe data types for point in time and duration.
- **ratio**
A type-safe fraction—that is, a number represented by an integer numerator and an equally integer denominator.
- **system_error**
Tools for portable error handling with error codes and exceptions.
- **<typeinfo>**
Runtime type information.
- **<functional>**
The `type` `function` is found in this header along with many predefined functor classes like `hash`, `bind`, `plus`, and `logical_and`.

In this chapter, you will find more classes and functions of the standard library.

28.1 “pair” and “tuple”

A `tuple` is something similar to an anonymous `struct`. Neither the `struct` itself nor its elements have names. You address the elements by their position. `pair` is older than `tuple` and encapsulates exactly two elements. These have the names `first` and `second`.

You will often find both in the standard library because they are so useful. It is possible to bundle multiple elements of different types together without much syntactic overhead.

28.1.1 Returning Multiple Values

Using a structure or class, you can also pack multiple pieces of data together and thus return multiple values from a function. In C++, however, there are special tools like `tuple` and `pair` to directly return multiple values of different types from a function using the `return` statement.

The use of `pair` is extremely simple. Primarily, you use `make_pair` to create.

```
// https://godbolt.org/z/4YM9d99xG
#include <iostream>
#include <string>
#include <vector>
#include <utility> // pair
using std::pair; using std::cout; using std::cin; using std::string;
std::vector<string> months { "Jan", "Feb", "Mar" };
std::vector<int> temps { 8, 12, 11 };
std::pair<string, int> monthWithTemp(size_t m) {
    auto month = months.at(m);
    auto temperature = temps.at(m);
    return std::make_pair(month, temperature);
}
int main() {
    std::pair data = monthWithTemp(1);
    cout << "Month : " << data.first << std::endl; // Output: Month : Feb
    cout << "Temperature : " << data.second << std::endl;
    // Output: Temperature : 12
}
```

Listing 28.1 How to return two values simultaneously using “pair”.

When declaring the `monthWithTemp` function, you can already tell from the return value that this function returns a `string` and an `int`. That's basically all you need to do. You just need to enter the two types (separated by a comma) between the angle brackets that should be returned. In `monthWithTemp(1)`, I call the function and also assign the return value to a `pair<string,int>`. The return is constructed from the two specified data using the `make_pair()` function and made into a `pair<string,int>`.

You can now access the individual values of `pair` using `first` (for the first value) and `second` (for the second value), as you can see in the example.

With `pair`, you are limited to exactly two elements. It is older than `tuple`, which is the generalization for any number of elements. You use it similarly.

```
// https://godbolt.org/z/5z4s6WMxG
#include <iostream>
#include <string>
#include <algorithm> // min, max
#include <tuple>
using std::tuple; using std::make_tuple; using std::get; using std::cout;
using std::string;
tuple<int,int,int> arrange(int a, int b, int c) {
    using std::min; using std::max;
    auto x = min(a,min(b,c));
    auto y = max(min(a,b), min(max(a,b),c));
    auto z = max(a,max(b,c));
    return make_tuple(x, y, z);
}
auto president(int year) {
    using namespace std::literals;
    if(year >= 2021)
        return std::make_tuple("Joseph"s, "Biden"s, "Democratic"s, 1942);
    if(year >= 2017)
        return std::make_tuple("Donald"s, "Trump"s, "Republican"s, 1946);
    if(year >= 2009)
        return std::make_tuple("Barack"s, "Obama"s, "Democratic"s, 1961);
    // Add more presidents as needed...
    return std::make_tuple("", "", "", 0); // Default case
}
int main() {
    tuple<int,int,int> zs = arrange(23, 42, 7);
    cout << get<0>(zs) << ' ' << get<1>(zs) << ' ' << get<2>(zs) << '\n';
    // Output: 7 23 42
    auto ps = president(2015);
    cout << get<1>(ps) << '\n'; // Output: Obama
}
```

Listing 28.2 With “tuple”, you can return any number of elements.

You combine multiple values of different types into a tuple using `make_tuple`. You can, as with `arrange` and `zs`, feel free to write out the tuple type. However, I recommend working with `auto`, as shown with `president` and `ps`. Here it is actually a `tuple<string, string, string, int>`, but you can save yourself the typing with `auto`.

Instead of accessing the elements with `first` and `second`, you now use `get<0>` and `get<1>`. For further elements, logically use larger numbers. Note that you can only specify fixed values (`constexpr`) as an index here. This does not work with variables.

You can even pass a type instead of a number, the type that you want.

```
// https://godbolt.org/z/TE1EcEYPv
// ... as before ...
int main() {
    tuple ps = president(2015);
    cout << get<int>(ps) << '\n';    // Output: 1940
    cout << get<string>(ps) << '\n'; // ✗ not unique
}
```

Listing 28.3 “get” works with a type as an index.

Because `int` appears only once in `tuple<string, string, string, int>`, you can use `get<int>` to access the corresponding element. `get<string>` does not work because that would not be unique.

“array” as “tuple”

The real array container (see [Chapter 24, Section 24.6.4](#)) for a fixed number of elements of the same type also offers the possibility to access its elements with `get<...>`. You can almost use an `array<int, 3>` like a `tuple<int, int, int>`.

If accessing elements in this way is too cumbersome for you, then bind the return value of a function to variables with `tie`.

```
// https://godbolt.org/z/zG7zYGf3K
// ... as before ...
int main() {
    using std::tie; using std::ignore;
    string lastName {};
    int birthYear {};
    tie(ignore, lastName, ignore, birthYear) = president(2015);
    cout << lastName << ' ' << birthYear << '\n'; // Output: Obama 1961
}
```

Listing 28.4 Decomposing tuples with “tie” and “ignore”.

When you use `tie` on the left side of an assignment as shown here, you can directly assign the individual parts of `tuple` and `pair` returns to previously declared variables. If you are only interested in parts of the return, then use `ignore` at the positions you are not interested in.

Structured Binding with “auto”

With C++17, it has become even easier to directly assign the returns of `tuple` or `array` to variables. This feature is called *structured binding*. This makes it possible to declare variables simultaneously:

```
auto [a, lastname, c, birthyear] = president(2015);
```

Here, four variables are automatically declared with the correct types and initialized with the corresponding parts of the returned tuple. For this, the expression on the right side needs a “tuple-like interface”; that is, it must support `get<>` and `tuple_size<>`, which is the case for `array`, `tuple`, and `pair`.

A C-array or a struct can also be on the right side.

`tuple` has overloaded operators for comparisons, particularly `<` and `==`, which are necessary for algorithms like `sort` and the standard containers. If the first elements of a tuple are equal, then the second elements are compared, and so on. This is called *lexicographical order*, because it works like sorting words in a dictionary.

```
// https://godbolt.org/z/sG4E66YhW
#include <iostream>
#include <string>
#include <tuple>
#include <vector>
#include <algorithm> // ranges::sort
using std::tuple; using std::get; using std::cout; using std::string;
int main() {
    std::vector<tuple<string,string,int>> armstrongs =
        { {"Armstrong", "Louis", 1901} // Initialize using initializer list
        , {"Armstrong", "Lance", 1971}
        , {"Armstrong", "Neil", 1930} };
    std::ranges::sort(armstrongs); // Lance < Louis < Neil
    for(const auto& a : armstrongs) {
        cout << get<0>(a) << ", " << get<1>(a) << ", " << get<2>(a) << "\n";
    }
}
```

Listing 28.5 Tuples implement lexicographical order.

First, you can see that it is easy to initialize a tuple when the type is given. This is the case here with `vector<tuple<string,string,int>>`. Therefore, it is sufficient to initialize each individual element with an initializer list of two `const char[]`s and an `int`.

When `sort` is performed, the tuples are compared with each other. The first “Armstrong” elements are always the same, and the comparison is not yet conclusive. Therefore, the second element is considered—for example, “Lance” and “Louis”.

Just as you can sort a `vector<tuple<...>>`, you can also use `tuple` as a key for ordered associative containers like `map<tuple<...>, ...>`.

The only thing needed to compare tuples is the comparability of the individual elements. This means that you can compare (and thus sort) `tuple<string,int,Ball>` exactly when you have implemented comparisons for `Ball`.

```
// https://godbolt.org/z/9qhMo6zbb
#include <iostream>
#include <string>
#include <tuple>
#include <map>
#include <unordered_set>
using std::tuple; using std::get; using std::cout; using std::string;

int main() {
    std::map<tuple<int,string>,double> m { {{12,"x"},3.14} };
    cout << m[{{12,"x"}}] << "\n"; // Output: 3.14
    std::unordered_set<tuple<int,string>> s { {12,"x"} }; // ⚡ no std::hash
}
```

Listing 28.6 Tuples as keys.

In an unordered associative container like `unordered_set`, you cannot use `tuple` without further precautions, as `std::hash(tuple<...>)` does not exist. However, this can be retrofitted if necessary.

The fact that `tuple` supports comparisons can also be used to simplify your own code. If you have your own data structure with multiple elements for which you want to implement a lexicographical order, you can rely on `tuple`. The `tie` function helps you here. It breaks down any `tuple` into its individual parts, which you can then associate with your own variables. The result is very compact code for lexicographical comparisons.

```
// https://godbolt.org/z/vdT7TTjWK
#include <iostream>
#include <string>
#include <tuple>
#include <map>
#include <unordered_set>
using std::tuple; using std::tie; using std::cout; using std::string;
struct Point {
    int x,y,z;
    bool operator<(const Point &b) {
        return tie(x,y,z) < tie(b.x, b.y, b.z);
    }
};
```

```
int main() {
    Point a{ 11, 22, 33 };
    Point b{ 11, 33, 0 };
    cout << std::boolalpha << (a < b) << "\n"; // Output: true
}
```

Listing 28.7 Forming tuples in-place using tie.

It is not difficult to implement a lexicographical order yourself. However, this is a tedious task into which errors often creep. When you use tuple with tie, the code is compact, clear, and correct.

Remember that you should not use make_tuple here as it would create copies of your data. tie works with references, so your data elements are compared without an additional copy.

The superlative of tie is forward_as_tuple, which you need when writing a function template that should assemble a tuple from multiple arguments while preserving rvalue references. This is a very advanced topic, which you will notice the first time you use it. You should not use forward_as_tuple until you have worked with std::forward at least once.

The last way to create a tuple is by using tuple_cat. You give tuple_cat a list of tuples and receive a single large tuple as a return, with all elements concatenated.

```
// https://godbolt.org/z/eq1dqWbe9
#include <iostream>
#include <string>
#include <tuple>
using std::tuple; using std::cout; using std::string;

int main() {
    tuple<int,string> a { 12, "gnorf" };
    tuple b { 666 };
    tuple<double,double,string> c { 77.77, 33.33, "frong" };
    tuple<int,string,int,double,double,string> r = std::tuple_cat( a, b, c );
    cout << std::get<2>(r) << "\n"; // Output: 666
}
```

Listing 28.8 Combining several small tuples into one large tuple.

Just as well—probably better—you could have written auto r = ... and let the compiler determine the correct type. Instead of tuple, you can also give tuple_cat a pair as an argument. As in most cases, the types are compatible.

Beware of “tuple” with References

You are not allowed to fill a standard container with references. There is a good reason for this: A `vector<T&>` would internally make assignments to references that do not do what you want.

However, `tuple<T&>` is allowed in principle because the standard library needs it to implement `tie`. I recommend that you do not use `tuple<T&>` yourself—and especially not to store references to data, which is treading on thin ice.

And the compiler will not stop you from using `vector<tuple<T&>>`, but here the ice you are treading on has already melted.

There are still a handful of useful tools for `tuple`. For example, you can determine the size of a `tuple` with `tuple_size<tuple<...>>::value`. If you ever “forget” the types of the elements, you can always get back to the individual types with `tuple_element<I,tuple<...>>::type`.

For concrete `tuple`, this is less interesting, but if you allow tuples as template parameters, then these *type traits* are useful helpers. Templates were discussed in detail in [Chapter 23](#).

```
// https://godbolt.org/z/Psdz9s9Gc
#include <iostream>
#include <string>
#include <tuple>
using namespace std;
template<typename Tuple>
auto back(Tuple &&tuple) {
    using Noref = typename remove_reference<Tuple>::type; // in case of Tuple&
    constexpr auto sz = tuple_size<Noref>::value;
    return get<sz-1>(forward<Tuple>(tuple));
}
int main() {
    tuple<string,int,string> enterprise = make_tuple("NCC", 1701, "D");
    cout << back(enterprise) << "\n"; // Output: D
}
```

Listing 28.9 Useful tuple type traits.

With `back`, you always get the last element of any `tuple`. Some other things are needed, but `tuple_size` is the core:

- As always, you use `get` to access a `tuple` element. `sz-1` is always the last element.
- `auto` automatically deduces the return type of the function. Without `auto`, this would be difficult but not impossible.

- You need `remove_reference` because `tuple_size` cannot handle references to `tuple`.
- `&&` in the parameter and `forward` ensure that this template works for values, references, and temp-values.

28.2 Regular Expressions

C++ now natively supports *regular expressions*. With them, you can search for patterns in strings and extract parts of them.

```
// https://godbolt.org/z/T1YMoM5as
#include <string>
#include <iostream>
#include <regex>
using std::regex; using std::sregex_iterator; using std::string;
const regex rgxMobile(R"(01[567]\d{6,10})"); // Mobile phone 0151-0179
bool isMobilephone(const string& text) {
    return std::regex_match(text, rgxMobile); // Does the text match completely?
}
bool containsMobilephone(const string &text) {
    return std::regex_search(text, rgxMobile); // somewhere in the text?
}
void listMobilephones(const string &text) {
    sregex_iterator begin{ text.cbegin(), text.cend(), rgxMobile };
    sregex_iterator end;
    for(auto it = begin; it != end; ++it)
        std::cout << it->str() << " "; // Matched text
} // "xyz01709999 abc 0161887766 uvw" -> "0161887766"
```

Listing 28.10 Matching, searching, and enumerating with regular expressions.

While `regex_match` checks if a text entirely matches a pattern, `regex_search` tests whether a pattern occurs anywhere within a larger text.

With various iterator classes like `sregex_iterator`, you can iterate over all matches of a pattern in a string.

German mobile phone numbers always start with 01, followed by one of the digits 5, 6, or 7. And they have a total length between 9 and 13 digits. The regular expression `R"(01[567]\d{6,10})"` catches them. We will discuss the pattern used for searching with `rgxMobile` in more detail later in [Table 28.1](#).

28.2.1 Matching and Searching

While `regex_match` always checks if the *entire* text matches the pattern, `regex_search` also finds matches contained anywhere in the text.

```
// https://godbolt.org/z/E9hcWGMW4
#include <regex>
#include <string>
#include <iostream>
using std::regex; using std::regex_match; using std::regex_search;
int main() {
    std::cout << std::boolalpha;
    regex pattern {"ello"};
    std::string text = "Hello world";
    auto b1 = regex_match (text.cbegin(), text.cend(), pattern); // doesn't match
    std::cout << b1 << "\n"; // Output: false
    auto b2 = regex_search(text.cbegin(), text.cend(), pattern); // found
    std::cout << b2 << "\n"; // Output: true
}
```

Listing 28.11 Search and match.

28.2.2 The Result and Parts of It

You can optionally pass a parameter to the search functions, in which the exact details of the result are stored. This is useful if you are interested in the start and end position of the match or even the covered subexpressions.

```
// https://godbolt.org/z/Yzefv59en
#include <regex>
#include <string>
#include <iostream>
using std::regex; using std::regex_search; using std::cmatch;
int main() {
    cmatch res; // for detailed results
    std::string text = "<h2>Result and parts of it</h2>";
    regex pattern{"<h(.)>([^\>]+)"}; // Search pattern with groups
    regex_search(text.c_str(), res, pattern); // Details to res
    std::cout << res[1] << ". " // ()-Group 1: H-Level
              << res[2] << std::endl; // ()-Group 2: H-Text
}
```

Listing 28.12 Accessing the match details.

This will print the following output:

2. Result and parts of it

The 2 is the heading level from <h2>, and the rest is the text between the angle brackets.

28.2.3 Found Replacement

It is just as easy to get a new string back in which the found pattern has been replaced with new text.

```
// https://godbolt.org/z/rErP43G89
#include <regex>
#include <string>
#include <iostream>
using std::string;
int main() {
    string text = "Title;Album;Artist";
    std::regex pattern(";");
    string new_str = std::regex_replace(text, pattern, string(","));
    std::cout << new_str << "\n"; // Output: Title,Album,Artist
}
```

Listing 28.13 Replace matches with new text.

In the new `new_str` string, all occurrences of the `;` pattern were replaced with `,`. If you only want to replace the first match, pass `regex_replace` and `regex_constants::format_first_only` as an additional parameter.

28.2.4 Rich in Variants

We have already seen that the text to be searched can be passed in different ways. The functions are indeed very flexible for various needs due to numerous overloads.

For example, there are variants of `regex_search` that could be informally summarized as follows:

- Text to be searched, such as `string`, `const char*`, or a pair of iterators
- Optional `match_results`, like `cmatch`
- The pattern to be searched as `regex`
- Optional flags, which default to `regex_constants::match_default` if omitted

The same applies to `regex_match` and `regex_replace` as well.

The standard library's regular expressions are not limited to character strings of `char`. Especially `wchar_t` and its string variant `wstring` are supported. In the background

general templates are working, the central one being `regex_traits`. This is where the functions for interpreting regular expressions and text are grouped together. Normally you do not need to do anything here, but there is nothing wrong with interpreting something other than `string` for a regular expression. For example, the standard definitions do not extend to new Unicode string variants `u16string` and `u32string`. Conceptually, however, there is nothing to prevent you from processing these strings; it may just require a bit of manual work when extending the templates.

I have stored the details of the search results in [Listing 28.12](#) in an instance of `cmatch`. But this is actually just a typedef for a template of `match_results` for searching in a string. For the other search types, there are other predefined typedefs:

- `smatch` for searching in `string`
- `cmatch` for searching in `const char*`
- `wsmatch` for searching in `wstring`
- `wcmatch` for searching in `const wchar_t*`

28.2.5 Iterators

You saw in [Listing 28.10](#) that besides searching, matching, and replacing, you can also iterate successively over all matches. This is done with the `regex_iterator` template class. We have already seen its work on `string` with the `sregex_iterator` specialization. For `char*`, `wstring`, and `wchar_t*`, there are typedef with `c`, `ws`, and `wc` instead of the prefix `s`.

Internally, `regex_iterator` uses the `regex_search` function for searching and is thus merely an adapter class for the iterators it provides. Therefore, they only point to corresponding `match_result` instances.

If you want to enumerate all matches in a string, it is recommended to use the iterator variants over a loop with `regex_match`. There are cases where you might otherwise produce an infinite loop; `regex_iterator` prevents this. Therefore, especially with regular expressions that could match the “empty string,” such as `"(abc)*"`, it is better to use the iterator variant.

28.2.6 Matches

Matches can do more than just output the found string with `str()`, as shown in [Listing 28.10](#).

You can use `length()` and `position()` to find out exactly where the match occurs in the text. The text *before* and *after* the match is provided by the call `prefix().str()` and `suffix().str()`. Subexpressions, as you have already seen in [Listing 28.12](#), can be obtained using `operator[]`.

28.2.7 Options

Regular expressions exist in the programming world in many different variants. If you do not specify otherwise, C++ supports the *ECMAScript* syntax—which is also known from JavaScript and is fundamentally related to Perl regular expression syntax. Anyone familiar with either should quickly find their way around in C++ as well. Other variants can be optionally supported by the implementation and are then selected via the constructor call:

```
regex rgx(R"...", regex_constants::Syntax);
```

Here, `Syntax` is a bitmask to select some different syntax variants. The standard specifies at least *ECMAScript*, *basic*, *extended*, *awk*, *grep*, and *egrep* as options. Implementations are free to offer more.

The bitmask can slightly modify the syntax with some modifiers:

- **icase**
Ignore case sensitivity.
- **nosubs**
Do not store subexpressions in the match result.
- **optimize**
Prefer match speed.
- **collate**
Consider the locale settings for expressions like `[a-z]`.

28.2.8 Speed

With the `optimize` option, note that the algorithm preprocesses the regular expression. This can take longer in some cases and consume a lot of memory. However, the result is faster when executing the search. In principle, how often the regular expression should be applied is a trade-off:

- If the expression is only passed to functions like `regex_search` or `regex_match` once or twice, the effort of `optimize` is not worth it.
- If you apply search and replace functions with a regular expression a dozen times or more, you can indeed save time overall.

In general, you don't need to worry about it. For an average expression, the difference is minimal. The case might occur with long, complex expressions that have many ambiguities and overlaps with themselves. An ambiguity would be, for example, `(abc|abd)`, because `ab` is matched by multiple subexpressions. A simple self-overlap occurs in `ababa`: with `abab`, a new possible match with `ab` also starts simultaneously.

Nevertheless, such cases are usually completely uncritical, and the algorithm can handle them very well, whether optimized or not. For gigantic, unmanageable, and unspecific expressions, optimization might take longer and require a lot of memory—but because it is then executed faster, the effort can still be worthwhile with multiple applications.¹

28.2.9 Standard Syntax, Slightly Shortened

Because the default is the ECMAScript regular expression syntax, I provide a brief summary of the most important elements of the syntax in [Table 28.1](#).

With `[class]`, you can describe a character from a group. You can enumerate characters like `[abcd]`, define ranges `[A-Za-z]`, invert `[^>]` (everything except `>`), and also use *class names* like `[[:alpha:]]`. Which class names are supported exactly is determined by the `regex_traits` used. But at least `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, and `w` exist. They correspond to the C function names, for example, `isalnum()`.

Pattern	Meaning	Pattern	Meaning
<code>ab</code>	a followed by b	<code>a{n,m}</code>	n to m times a
<code>a b</code>	Alternative	<code>()</code>	Grouping
<code>.</code>	Any character	<code>^ \$</code>	Start and end of string
<code>a?</code>	0 or 1 time a	<code>[class]</code>	Character from class
<code>a*</code>	0 or more times	<code>\d \w \s</code>	Class shortcuts
<code>a+</code>	1 or more times	<code>\b</code>	Word boundary

Table 28.1 A very brief overview of ECMAScript regex syntax.

You can also combine and negate enumerations and class names—for example:

- `[[:alpha:]]` matches any letter.
- `[abc[:digit:]]` matches the letters a, b, c and any digit.
- `[^[:space:]]` matches any character that is not whitespace.

There are abbreviations for some class names. For example, you can write `\d` instead of `[[:digit:]]`. You can see abbreviations in detail in [Table 28.2](#). The `w` in `\w` stands for *word*, meaning letters that can appear in a word.

¹ The algorithm converts the expression into a deterministic finite automaton (DFA) during optimization. In the worst case, this can require exponentially more time and memory.

Abbreviation	Stands For	Abbreviation	Stands For
\d	<code>[[:digit:]]</code>	\D	<code>[^[:digit:]]</code>
\s	<code>[[:space:]]</code>	\S	<code>[^[:space:]]</code>
\w	<code>[_[:alnum:]]</code>	\W	<code>[^_[:alnum:]]</code>

Table 28.2 Abbreviations for character classes.

To support international texts, there are specially designated group names. There are *collation groups* (*collations*) like `[[:span-ll:]]`. This allows `ll` to be matched as a single character in Spanish. With *equivalence groups* like `[[:e=]]`, an `e` and all its accented variants, `è`, `é`, and `ê`, can be matched in French.

28.2.10 Notes on Regular Expressions in C++

Differences in Algorithm

For most applications, the subtleties of regular expression searching are probably irrelevant. Will a short string be found first, or the one further up? Those who are more used to Perl or POSIX expressions might be confused, so we want to point out the small difference here.

Comparing the results of a search according to the POSIX standard and the ECMAScript standard yields the results shown in [Table 28.3](#).

Search Pattern	Text	POSIX	ECMAScript
<code>a ab</code>	"xaby"	"ab"	"a"
<code>.*(a xayy)</code>	"zzxayyzz"	"zzxayy"	"zzxa"
<code>([:alnum:])</code>	" abc def xyz "	<code>m[1] == "abc"</code>	<code>m[1] == "z"</code>

Table 28.3 Differences between POSIX and ECMAScript in regular expressions.

The results may therefore differ slightly:

- **ECMAScript**
Prefers shorter matches if they are the first to return a match.
- **POSIX**
If in doubt, returns a longer hit that starts further left.

With the advanced search options available in Perl, neither flavor can compete. Compared to POSIX, the ECMAScript variant offers the advantage that it often allows writing a more intuitive search pattern. Who hasn't tried to find an opening element in an HTML text with a regex that matches the closing element, such as `<font>...</font>?`

The attempt to achieve this with the `<font[^>]*>.*</font>` pattern fails in both cases because `.*` consumes as much text as possible. Not only would the matching elements be found, but everything up to the last `</font>` element in the text—including additional `<font>` and `</font>`.

In ECMAScript syntax, compared to POSIX (for example), you can make an otherwise “greedy” part of the expression nongreedy by appending a `?`. Thus, the `<font[^>]*>.*?</font>` pattern accomplishes the task. In case of doubt, `.*` will now consume text that is as short as possible.

Readable Regular Expressions

I use a lot of regular expressions in my daily life, and you would probably be amazed at how often and for what. They are powerful tools, but they also have their pitfalls. In my experience, you should *always* outsource a search, match, or replace algorithm into an easily testable function and then feed it with a meaningful set of test cases in the unit test. More than usual, you can overlook one thing or another with regular expressions.

So fine, you have written your function, and it contains a great regular expression. And you have tests—wow! But in three weeks, the trickiest pitfall will catch up with you: you no longer understand the expression. You want to extend or correct it, but no matter what you change, the algorithm just breaks.

Yes, regular expressions are easy to write once you get the hang of them. But reading and understanding them again often becomes difficult. They are not a prime example of meeting the guideline that code should be self-documenting and require few to no comments.

But there is a solution—there are several, in fact. I advise you to write your regular expressions to be readable from the start.

Assume you have numerous messages like this:

```
score 400 for 2 nights at Minas Tirith Airport
```

Here, the texts and numbers can change. You are supposed to extract the numbers 400 and 2 as well as the hotel name, Minas Tirith Airport, after `at`. The regular expression can look like this:

```
const string pattern = R"(^score\s+(\d+)\s+for\s+(\d+)\s+nights?\s+at\s+(.*)");
```

In the program, it looks like the following listing.

```
// https://godbolt.org/z/dG5vE3heM
#include <regex>
#include <string>
#include <iostream>
using std::regex; using std::regex_search; using std::cmatch;
```

```

const regex pattern{R"(^score\s+(\d+)\s+for\s+(\d+)\s+nights?\s+at\s+(.*)")};
void extract(const std::string &text) {
    cmatch res;
    regex_search(text.c_str(), res, pattern);
    std::cout << res[1] << ", " << res[2] << ", " << res[3] << "\n";
}

int main() {
    extract("score 400 for 2 nights at Minas Tirith Airport");
    extract("score 84 for 1 night at Prancing Pony");
}

```

Listing 28.14 Hard to maintain regular expression.

This is still barely readable. To improve readability, as always, there are at least three options:

- Restructure
- Use descriptive names
- Comment

Combining the first two approaches, you could perhaps break down the large regex into several smaller ones, give the parts names, and then combine them.

```

// https://godbolt.org/z/TK6n4dG37
const string scoreKeyword = R"(^score\s+)";
const string numberOfPoints = R"((\d+))";
const string forKeyword = R"(\s+for\s+)";
const string numberOfNights = R"((\d+))";
const string nightsAtKeyword = R"(\s+nights?\s+at\s+)";
const string hotelName = R"((.*))";
const regex pattern{ scoreKeyword + numberOfPoints +
    forKeyword + numberOfNights + nightsAtKeyword + hotelName };
}

```

Listing 28.15 Name parts and then combine them.

Perhaps this is more maintainable. At least this approach is recommended if parts of the expression might be used elsewhere in the program.

The third approach to commenting also works, except that C++ comments within the expression—within the string—do not work.

Here you can use the feature that C++ directly concatenates adjacent string literals if only whitespaces remain after removing comments. In this way, you can then comment parts of the expression in an almost “embedded” way.

```
// https://godbolt.org/z/MszTjE3dv
const regex pattern{R"^(score)"
    R"(\s+)"
    R"((\d+))"           // points
    R"(\s+)"
    R"(for)"
    R"(\s+)"
    R"((\d+))"           // number of nights
    R"(\s+)"
    R"(night)"
    R"(s?)"              // optional: plural
    R"(\s+)"
    R"(at)"
    R"(\s+)"
    R"((.*))"            // hotel name
    R"()"};
```

Listing 28.16 Commenting within the expression.

All raw string fragments `R"(...)"` are directly adjacent, ignoring whitespaces and comments. The C++ preprocessor combines these into a single string literal.

28.3 Randomness

True random numbers are a difficult thing for computers. But in some applications, you need good random numbers. And the good old `rand()` from `<cstdlib>` no longer suffices—or `<stdlib.h>`, as the header is called in the C11 standard.² Even in the standard document, you can read about `rand()`: “There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.” Or, in short: If you value randomness, do not use `rand()`! Instead, use `<random>`, because according to Bjarne Stroustrup, it is “what every random number library wants to be when it grows up.”³

In C++, the `<random>` header provides everything you need for random number generation. For 99% of all applications, the provided functionalities should suffice. You may build upon the classes and add your own extensions if necessary. However, let me also

² *Programming Language—C*, ISO/IEC, <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf>, N1570, ISO/IEC 9899:201x

³ *C++11 - the new ISO C++ standard*, <https://stroustrup.com/C++11FAQ.html#:~:text=%60what%20every%20random%20number%20library%20wants%20to%20be>, Bjarne Stroustrup, 2016-08-19, [2024-08-15]

mention that `<random>` does not cater to the 1% of applications that deal with cryptography. For these applications, you will need third-party libraries, and you can use `<random>` as an inspiration.

If you just want to simulate a die with many sides, it is not really complicated. But if you are a physicist or mathematician and know that you need a random number with a *student distribution* at a certain performance, then you need to dig deeper, but you will still find what you need in `<random>`.

This part of the library offers something for every user level:

- **Occasional users**

A preconfigured generator and two uniform distributions

- **Intermediate and advanced users**

Nine more generators, 18 more distributions, a true random number generator, and a helper class for seed values

- **Experts**

Six configurable adapters for combining generators and a function template for writing custom distributions

28.3.1 Rolling a Die

For average users who want to get a uniformly distributed random number, the next example looks simple.

```
// https://godbolt.org/z/ocP83a7ac
#include <random>
#include <vector>
#include <iostream>
void rollDice() {
    std::default_random_engine engine{};           // Normal quality randomness
    std::vector<size_t> counts{0,0,0,0,0,0};
    std::uniform_int_distribution<int> d6{0, 5}; // uniformly distributed integers
    for(auto i=1200*1000; i>0; --i)
        ++counts[d6(engine)];
    for(auto c : counts) std::cout<<" "<<c;
    std::cout << '\n';
}
int main() {
    rollDice();
}
```

Listing 28.17 This is how you get a uniformly distributed random number between two bounds.

The output is the number of times each number was rolled. If all numbers were rolled equally often, you would see six times 200000. But because it is rolled, it is instead this:

```
199664 200248 199727 200143 200373 199845
```

As you can see, generating random numbers now consists of two things:

- **Engine**

The “engine” is responsible for generating the (possibly pseudo)random bits. Different generators produce random numbers with different qualities and at different speeds.

- **Distribution**

The distribution is responsible for mapping the bit representation to the actual numbers to be generated. These can be uniformly distributed, within a certain range, clustered at specific points, and so on.

If you combine these two things, you get the desired random number generator. A distribution like `d6` defines the `operator()` call operator. If you pass an engine as an argument to it, you get a random number of the desired distribution with the desired quality, as per the preceding example:

```
auto number = d6(engine);
```

And `number` is now something between 0 and 5 (inclusive) with equal probability. If you don't like having to always carry and specify the engine here, simply combine both into a function object—for example, with a lambda:

```
auto d = [&]() { return d6(engine); };  
std::cout << d() << '\n';
```

And when I run [Listing 28.17](#) again, I get the same output. And again and again—but that's a *very* unlikely coincidence, isn't it? It would be indeed, but all engines (except one, which we will get to shortly) are *pseudorandom number generators*: initialized with the same seed, they always produce the same bit sequence—which looks random under certain considerations and is sufficient for many (most?) applications. The reason I always got the exact same sequence of pseudorandom numbers here is that I always chose the same *seed* for the engine—namely, the default value. You can pass a different seed as a constructor argument:

```
std::default_random_engine engine{42};  
std::default_random_engine engine{886699};  
std::default_random_engine engine{time(0)}; //from <ctime>
```

Each seed value provides a different pseudo-random sequence. If you start with a variable value—like here with `time(0)`, the current time—you will always get a different sequence. This is sufficient for standard cases. If you need more, there is the `std::seed_seq` helper class.

However, it may be that you get a different sequence of pseudorandom numbers on your machine than I do. Even though the generators are all well-defined by the standard, the `default_random_engine` is mapped to one of the generators by your specific implementation; which one is decided by the manufacturer. If you need true reproducibility, set both the generator *and* the seed value.⁴

28.3.2 True Randomness

True random numbers are provided by the `random_device` generator. This does not derive randomness from a seed and a mathematical formula but from various true randomness-producing components of your computer, such as mouse movements or hard disk head movements. However, this generator has a few significant disadvantages:

- It is slow. Especially on server systems (without a mouse and the like), it is sometimes *very* slow.
- Results are not reproducible, for example, for debugging; thus, the advantage can turn into a disadvantage.
- The C++ implementation does not have to offer this engine; it is optional.

You can check with the `entropy()` function of `random_device` whether `random_device` can really generate random numbers. If you see 0.0, then some other—deterministic—engine is used.

28.3.3 Other Generators

An engine implements `operator()`, which returns a random raw bit sequence. You can't do much with it for real experiments yet. Only when combined with a *distribution* does it become a random number in the correct range.

The *distribution*—like `uniform_int_distribution<int> d6{0,5}`—also overloads the `operator()`. If you pass the engine as a parameter to it, the bit sequence becomes an appropriate random number—for `d6` in the range between 0 and 5 (inclusive)—with each engine.

Engines differ in the quality of the bit sequence they return. This quality is mathematically calculable and extremely important in some scenarios. Our deterministic computers have difficulties with true randomness. Therefore, the algorithms only generate *seeming* randomness, so-called pseudorandom numbers.

As a rule of thumb, the sequence of random numbers gets better the more time-consuming the generation of the bits is. I have written a program here that first allows you

⁴ However, this only applies at the generator level—that is, for the generated bits. How exactly the distribution folds these bits is again left to the manufacturer.

to measure the speed of the individual engines and second shows all the engines that might be present on current systems.

```
// https://godbolt.org/z/e67356bMq
#include <random>
#include <chrono> // system_clock
#include <format>
#include <iostream>

using namespace std;
using namespace chrono;
const size_t LOOPS = 10*1000*1000;

template<typename ENGINE>
void measure(const string &title, ENGINE &engine) {
    const auto start = steady_clock::now();
    /* work hard */
    unsigned long long checksum = 0;
    size_t loops = LOOPS;
    while(loops-- > 0)
        checksum += engine();
    const auto now = steady_clock::now();
    nanoseconds dur_ns = now - start;
    cout << std::format(" {}: {:5} ns/loop {:12} ns\n",
        title, dur_ns.count() / LOOPS, dur_ns.count());
}

int main() {
    { default_random_engine e{}; measure(" default", e ); }
    { random_device e{}; measure(" device", e ); }
    { minstd_rand0 e{}; measure(" minstd_rand0", e ); }
    { minstd_rand e{}; measure(" minstd_rand ", e ); }
    { mt19937 e{}; measure(" mt19937 ", e ); }
    { mt19937_64 e{}; measure(" mt19937_64", e ); }
    { ranlux24_base e{}; measure(" ranlux24_base", e ); }
    { ranlux48_base e{}; measure(" ranlux48_base", e ); }
    { ranlux24 e{}; measure(" ranlux24 ", e ); }
    { ranlux48 e{}; measure(" ranlux48 ", e ); }
    { knuth_b e{}; measure(" knuth_b", e ); }
    {
        using wide_t = unsigned long long ;
        independent_bits_engine<ranlux48, sizeof(wide_t)*8, wide_t> e{};
        measure("indep<ranlux> ", e );
    }
}
```

```

{
    using wide_t = unsigned long long;
    independent_bits_engine<
        default_random_engine,
        sizeof(wide_t)*8, wide_t>
        e {};
    measure("indep<default>", e );
}
}

```

Listing 28.18 Speeds of the random number generators.

The results always vary on different systems, of course. Consider the following excerpt:

default:	12 ns/loop	126342116 ns
device:	41 ns/loop	417456044 ns
minstd_rand0:	12 ns/loop	122354003 ns
indep<ranlux>:	1086 ns/loop	10865420582 ns

As you can see, the device here (!) is not that slow, and if you use `indep<ranlux>`, you should have a reason for it.

28.3.4 Distributions

Random numbers can be generated with various characteristics or *distributions*—from a simple six-sided die to floating-point values.

With a six-sided die, each of the numbers 1 to 6 should appear roughly equally often. But other random experiments behave differently. If you throw 10 coins (together) and count how often heads appears, then the possible outcomes 0 to 10 are binomially distributed. If you repeat this experiment several times in a row, you will see 0 heads less often than 5 heads. For such experiments, you do not need to generate the numbers 0 or 1 for heads or tails 10 times from the computer, but you can directly generate a number between 0 and 10 with the appropriate probability using `binomial_distribution`.

You have already seen in [Listing 28.17](#) what a uniform distribution looks like for a single die. Let's choose a binomial distribution in [Listing 28.19](#) to simulate tossing 10 coins.

```

// https://godbolt.org/z/rv5fWVP5Ya
#include <random>
#include <vector>
#include <iostream>
using namespace std;
int main() {
    static const size_t size = 10;

```

```
default_random_engine e{};           // Random number generator
vector<size_t> counts(size+1);
binomial_distribution<int> coins{size}; // tosses 10 coins, 0 to 10 heads
for(auto i=120*1000; i>0; --i)
    ++counts[coins(e)];               // Tossing coins
for(auto c : counts) cout<<" "<<c;
cout << '\n';
// Example output:
// 109 1159 5344 14043 24806 29505 24544 13973 5252 1150 115
}
```

Listing 28.19 A binomial distribution.

As you can see, 109 is much less likely to get 0 heads than 29505 is to get 6 heads—a binomial distribution.

Suppose you wanted a uniform distribution, but this time not of `int` values, but instead of `double`. You need a different distribution function—namely, `uniform_real_distribution<double>`.

```
// https://godbolt.org/z/94McsP7PG
#include <random>
#include <iostream>

int main() {
    std::default_random_engine e{};
    std::uniform_real_distribution<double> unif{3,7}; // in the half-open interval [3,7)
    double u = unif(e);                               // Generate random number
    std::cout << u << '\n';                          // Example output: 3.52615
}
```

Listing 28.20 A “double” random number.

Different Roll Every Time

If you already provide the range to the constructor of `uniform_int_distribution<int>` where the random numbers should be generated, do you always need to create a new distribution if the boundaries change each time?

No, fortunately not. Each distribution has a `param_type` with the same constructor arguments as the distribution. You can pass such a `param_type` instance to the generating call of `operator()` and thus change the conditions for generation from call to call.

```
// https://godbolt.org/z/xz64qK3e5
#include <random>
#include <iostream>
```

```

int main() {
    std::default_random_engine e{};
    using Dstr = std::uniform_int_distribution<int>;
    Dstr card{}; // generate distribution
    for(int n=32; n>=1; --n)
        std::cout << " "<< card(e, Dstr::param_type{1,n} ); // parameters only here
    std::cout << '\n';
    // output for example:
    // 15 23 14 15 6 2 17 17 22 9 11 17 11 10 11 16 16 8 6 9 7 4 14 2 3 2 1
}

```

Listing 28.21 Change generation parameters for each random number individually.

Here I simulate that the deck of cards gets one card smaller with each draw. The card number in the ever-decreasing deck is printed.

Distributions over Distributions

The exact formulas of all distributions and what their parameterizations mean are, of course, mentioned in the standard. To the mathematically inclined, the names will probably mean more. For those, the parameterizations of the distribution functions are probably of more interest. In [Table 28.4](#), *real* stands for a real number type such as float or double. An *int* stands for an integer type, such as int, but of course others like long long are also possible.

Distribution	Distribution Parameters
Uniform Distributions	
uniform_int_distribution	Int a = 0, Int b = max()
uniform_real_distribution	Real a = 0.0, Real b = 1.0
Bernoulli Family	
bernoulli_distribution	double p = 0.5
binomial_distribution	Int t = 1, double p = 0.5
geometric_distribution	double p = 0.5
negative_binomial_distribution	Int k = 1, double p = 0.5
Poisson Family	
poisson_distribution	double mean = 1.0
exponential_distribution	Real lambda = 1.0
gamma_distribution	Real alpha = 1.0, Real beta = 1.0

Table 28.4 Parameters of the distribution functions.

Distribution	Distribution Parameters
weibull_distribution	Real a = 1.0, Real b = 1.0
extreme_value_distribution	Real a = 0.0, Real b = 1.0
Normal Distributions	
normal_distribution	Real mean = 0.0, Real stddev = 1.0
lognormal_distribution	Real m = 0.0, Real s = 1.0
chi_squared_distribution	Real n = 1
cauchy_distribution	Real a = 0.0, Real b = 1.0
fisher_f_distribution	Real m = 1, Real n = 1
student_t_distribution	Real n = 1
Sample Family	
discrete_distribution	initializer_list<double> wl
piecewise_constant_distribution	initializer_list<R> bl, Op1 fw
piecewise_linear_distribution	initializer_list<R> bl, Op1 fw

Table 28.4 Parameters of the distribution functions. (Cont.)

This is the list of distributions specified in the standard. It is up to the library manufacturer to contribute more. g++ 6.2, for example, adds the following:

- arcsine_distribution
- beta_distribution
- exponential_distribution
- hoyt_distribution
- hypergeometric_distribution
- k_distribution
- logistic_distribution
- nakagami_distribution
- normal_mv_distribution
- pareto_distribution
- rice_distribution
- triangular_distribution
- uniform_on_sphere_distribution
- von_mises_distribution

28.4 Mathematical

28.4.1 Fraction and Time: “<ratio>” and “<chrono>”

To work with time, you need the <chrono> header. Everything in it is in its own namespace, `std::chrono`.

`std::chrono` introduces central time units for seconds and milliseconds, but also hours and days. All of this is heavily based on compile-time fractions that come from the <ratio> header. Why? Because, for example, one second is the same as 1/86,400 day—and the compiler automatically converts things from <ratio> and <chrono> for you. <chrono> uses <ratio> as a foundation, and you could say it represents an example usage.

<chrono> contains types and functions for the following aspects:

- *Time spans*, the difference between points in time
- *Points in time* for relative comparison to obtain time spans
- *Clocks* with different reference times such as the computer's system time or GPS
- *Calendars* for date calculation and representation
- *Time zones* for converting clocks and calendars between different locations around the world

Motivation

Imagine you saw the following line in the code:

```
sleep(10);    // ⚡ what unit is meant?
```

What does it tell you? It waits. How long? Ten milliseconds? Ten seconds? Experience suggests that one of the two is likely; less likely is 10 nanoseconds or 10 minutes. But according to the rule that source code should be self-explanatory and ideally without comments, it is possible with <chrono> to write the following:

```
sleep(10s);
```

This is clear and not ambiguous. The `s` is a suffix defined by `operator""` for the literal 10, as you could also define it yourself (see [Chapter 23, Section 23.7](#)), except that your suffixes must start with an underscore `_`. The standard library does not have this restriction, and thus 10s in `sleep` generates the following code:

```
sleep(std::chrono::seconds{10});
```

This in turn stands for the following:

```
sleep(std::chrono::duration<unsigned long, std::ratio<1>>{10});
```

The `std::ratio<1>` stands for the *seconds* unit. If you want to specify 10 minutes, you don't need to calculate it yourself but can instead make one of the following calls:

```
// https://godbolt.org/z/173nr4xxd
sleep(10min);
sleep(std::chrono::minutes{10});
sleep(std::chrono::duration<unsigned long, std::ratio<60>>{10});
```

There is another great side effect of this strong type system. When you define the `sleep` function, you naturally specify that the parameter should be a duration—for example:

```
// https://godbolt.org/z/5snr1hq9b
#include <chrono>
void sleep(std::chrono::seconds s) { // takes seconds as duration
    /* ... */
}
/* ... */
int main() {
    using namespace std::chrono; // make literals available
    sleep(10min); // wait for 10 minutes, i.e., 600 seconds
    sleep(10ms); // ✗ 10 milliseconds? Cannot be represented with seconds.
}
```

When you define `sleep` to expect seconds as a parameter, users can still provide `10min` as a parameter, and `<chrono>` will automatically perform the conversion.

And `<chrono>` also prevents someone from trying to specify `10ms` as a time span. If you define `sleep` to take seconds as a parameter, there must be a reason for that. `sleep` will not be able to work in the millisecond range. So by choosing the parameter type for `sleep`, you are also making a statement about the granularity with which your `sleep` operates—and you do so without extra comments or documentation.

And what if you find that you need to refine your granularity to nanoseconds? Then you do *not* need to go through the source code and modify all `sleep` calls, nor do you need to introduce a `sleepNanoseconds` function. You only change the `sleep` function:

```
void sleep(std::chrono::nanoseconds ns) { // takes nanoseconds as duration
    /* ... */
}
```

And `<chrono>` automatically handles the conversion at all calling sites at compile time.

“`chrono::seconds`”

`chrono::duration` stands for *duration*, for a *time span*. This can be something like three seconds, three minutes, three hours, and so on.

The basis of `<chrono>` is the seconds in the form of `std::chrono::seconds`. At runtime, an instance of this type is no different from an `int64_t` or `long long`; both occupy eight bytes. So you can (trivially) move, copy, and construct it, and it costs the computer almost no time.

At its core, `seconds` is a very simple type, and you can roughly think of it like this:

```
// https://godbolt.org/z/h8vsjxsME
#include <cinttypes> // int64_t
namespace std { namespace chrono {
class seconds {
    int64_t sec_;
public:
    seconds() = default;
    // ... etc ...
};
} }
```

That means you can initialize `seconds` like this, for example:

```
std::chrono::seconds s1;    // uninitialized value
std::chrono::seconds s2{};  // zero initialized
```

What you cannot do is this:

```
std::chrono::seconds s = 3; // ✗ Error: not implicitly from int
```

This is very important, because just as important as what you can do with `<chrono>` is what you *cannot* do with it. For example, if this initialization were allowed, you would be able to write `sleep(3)` again, and you would have automatic conversion of `int` to `seconds` (or `nanoseconds`?).

Instead, you need to write this:

```
std::chrono::seconds s{3};
```

This way, you protect yourself from errors.

You can output time spans like these to a stream. `cout << s`, for example, outputs `3s`. This is possible since C++20. Before that, you would use `count()` to get the enclosed value:

```
std::cout << s.count() << "s\n";
```

Using `chrono::duration` as a parameter or variable looks like the following listing in practice.


```
// https://godbolt.org/z/47j8h9e7v
#include <chrono>
#include <iostream>
void sleep(std::chrono::seconds dur) {
    std::cout << dur.count() << "s\n";
    /* ... */
}
int main() {
    using namespace std::chrono;
    sleep(3);           // ✗ Error: no implicit conversion from int
    sleep(seconds{4}); // okay
    sleep(5s);          // okay
    seconds x{6};
    sleep(x);           // okay
    auto y = 10s;
    y += 20s;           // Incrementing with seconds
    sleep(y);           // now 30s
    y = y - 6s;         // Subtraction of seconds
    sleep(y);           // and now only 24s
    y /= 2;             // Division by a scalar
    sleep(y);           // 12s
    sleep(y + 7);       // ✗ Error: seconds+int is not allowed
}
```

Listing 28.22 With “seconds”, you can perform calculations.

As you can see, you can perform reasonable calculations with `seconds`. Various arithmetic operators are overloaded. Can you perform addition of seconds? Yes. Division by a scalar? Yes. Addition of a scalar? No. This is because with `y+7` you again do not know whether seven seconds or seven years is meant, and `<chrono>` protects you from this ambiguity.

In addition to arithmetic operations, you can also perform comparisons.

```
// https://godbolt.org/z/8c1r5xrs5
#include <chrono>
#include <iostream>
using std::chrono::operator""s; // only make literal suffix available
constexpr auto limit = 10s;
void action(std::chrono::seconds dur) {
    if(dur <= limit) { // compare
        std::cout << dur.count() << "s\n";
    } else {
        std::cout << "too long!\n";
    }
}
```

```
int main() {
    action(4s);    // Output: 4s
    action(20s);  // Output: too long!
}
```

Listing 28.23 You can compare “seconds”.

For seconds, a variety of meaningful arithmetic and comparison operators are defined, and many nonmeaningful ones are intentionally not defined.

In [Table 28.5](#), I have listed the methods of seconds for you.

Method	Description
<code>count()</code>	Access to the enclosed value
<code>min()</code>	static, largest possible negative representable duration
<code>max()</code>	static, largest possible representable duration
<code>zero()</code>	static, zero duration

Table 28.5 Methods of seconds.

Because seconds internally uses `int64_t`, you can cover a range of nearly 600 billion years between `seconds::min()` and `seconds::max()`.⁵

Other Time Spans: “`std::chrono::duration`”

However, you not only have seconds as a time unit available, but also other time units as mentioned at the beginning. The standard specifies at least those in [Table 28.6](#). The implementation may offer more. The types from days to years were newly added in C++20, but they do not have suffixes. The `d` and `y` suffixes define a day and year of a calendar, not the duration of days or years.

Type	Seconds	Suffix
years	<code>ratio<31556952,1></code>	
months	<code>ratio<2629746,1></code>	
weeks	<code>ratio<604800,1></code>	
days	<code>ratio<86400,1></code>	

Table 28.6 Various “duration” types of the standard library.

⁵ The exact range is not specified by the standard library, but in practice this order of magnitude should be achieved.

Type	Seconds	Suffix
hours	ratio<3600,1>	h
minutes	ratio<60,1>	min
seconds	ratio<1,1>	s
milliseconds	ratio<1,1000>	ms
microseconds	ratio<1,1000000>	us
nanoseconds	ratio<1,1000000000>	ns

Table 28.6 Various “duration” types of the standard library. (Cont.)

By using `ratio` as the basis of these time units, `<chrono>` allows the automatic conversion of all these time units into each other.

```
// https://godbolt.org/z/qrs63vdz1
#include <chrono>
#include <iostream>
int main() {
    using namespace std::chrono;           // Allow suffixes
    seconds mySecs = 121s;                  // seconds{121}
    std::cout << mySecs.count() << "s\n";   // Output: 121s
    milliseconds myMillis = mySecs;         // automatically converted
    std::cout << myMillis.count() << "ms\n"; // Output: 121000ms
    nanoseconds myNanos = mySecs;
    std::cout << myNanos.count() << "ns\n";  // Output: 121000000000ns
    minutes myMinutesErr = mySecs;          // ⚡ Error: Conversion with loss
    minutes myMinutes = duration_cast<minutes>(mySecs); // explicit works
    std::cout<<myMinutes.count()<<"min\n";  // Output: 2min
}
```

Listing 28.24 Automatic conversion between time units.

As you can see, you then get the converted size in `count()`. So long as the conversion can take place without loss, the transformation works automatically. It would also work implicitly for parameters and returns, as you have already seen in the introductory example with `sleep(...)`.

It is different when information would be lost during the conversion. This would be the case, for example, if you converted `mySecs` to `minutes`. Then, 121 seconds would become 2 minutes, and the remaining 1 second would be lost. Here, the compiler complains if you want to do it implicitly.

If you know what you're doing, you can force exactly this conversion with `duration_cast<...>`. The result is a rounding toward zero (decimal places are simply truncated). You get other rounding approaches with the `floor`, `round`, and `ceil` functions.

To be clear, `duration_cast<>` is not built-in like `static_cast<>` and its kin, but is a function template that is fully named `std::chrono::duration_cast<>` and is part of `<chrono>`.

All this works because `<chrono>` knows the relationships between different time units and knows whether and how to convert them into each other.

Do Not Convert Yourself

When you start working with `<chrono>`, let the library handle the conversions. Do not convert yourself, like in the following listing.

```
// https://godbolt.org/z/Y3dnsMYcs
#include <chrono>
#include <iostream>
int main() {
    using namespace std::chrono;
    milliseconds myMs = 4'000ms;
    std::cout << myMs.count()/1000 << "s\n"; // ⚡ here loss threatens
    std::cout << myMs.count()*1000 << "us\n"; // ⚡ here loss threatens
    std::cout << duration_cast<seconds>(myMs).count() << "s\n"; // explicit
    std::cout << microseconds(myMs).count() << "us\n"; // like implicit
}
```

Listing 28.25 Do not make your own calculations using “count()”.

It is much less error-prone to let `<chrono>` do the work in one place. The performance is also not worse than if you programmed it yourself by hand.

If the compiler tells you somewhere that it cannot convert something, believe it. Do not bypass its warning by using `count()` and your own calculation. The probability is high that you have made a logical error somewhere.

You can also mix conversions and arithmetic. An expression can contain different units. So long as no information is lost during the conversions, it works seamlessly.

```
// https://godbolt.org/z/vTx4cdqzq
#include <chrono>
#include <iostream>
void showDuration(std::chrono::milliseconds dur) {
    std::cout << dur.count() << "ms\n";
}
int main() {
    using namespace std::chrono;
```

```
auto x = 2s;
auto y = 3ms;
showDuration(x + y); // Output: 2003ms
showDuration(x - y); // Output: 1997ms
}
```

Listing 28.26 You can mix units.

Here, `x` is automatically converted to milliseconds before `+` or `-` is performed.

Custom “duration” Types

If you need other time units (Femtoseconds? Ten-day spans?) or if the 64 bits of `second` consume too much memory, you can easily create a new type that works seamlessly with the others.

```
// https://godbolt.org/z/xMz4s8K8K
#include <chrono>
#include <iostream>
using namespace std::chrono; using std::cout;
using seconds32 = duration<int32_t>; // other representation
using ten_day = duration<int, std::ratio<86400*10>>; // other time unit
using fseconds = duration<double>; // Floating point representation
int main() {
    seconds32 s{5};
    cout << milliseconds(s).count() << "ms\n";
    ten_day z{1};
    hours h = z; // Conversion free
    cout << "1 ten_day has "<< h.count() << " hours\n"; // ...240...
    fseconds fs{23.75};
    cout << fs.count() << "s\n"; // Floating point output
    auto printDur = [](fseconds f) { cout << f.count() << "s\n"; }; // Function
    printDur(45ms + 63us); // Conversion to fseconds
    // Output: 0.045063s
}
```

Listing 28.27 Create new time units or more efficient representations.

Conversions between the `seconds32` and `week` types and to the normal time types work easily.

You will receive special treatment if you choose a floating-point representation for your own time type, in which case `<chrono>` assumes that no precision is lost during conversions. The expression `45ms + 63us` is implicitly converted to microseconds for the calculation. The `printDur` function takes `fseconds` as parameter. The compiler allows the conversion of microseconds and `fseconds` and performs the conversion correctly. This

would not work with the integer-based `seconds` because the compiler would protect against loss of precision. If you use a float type, the compiler allows the conversion. An explicit conversion with `duration_cast<>` then is not necessary.

Calendars, Clocks, and Time Zones

For C++20, the `<chrono>` header has been extensively expanded. While the C++11 version only included durations, time points, and a few clocks, it now includes calendars, leap seconds, time zones, and additional clocks as well as formatting and parsing.

To work with calendars in a type-safe manner, you now have access to the calendar types shown in [Table 28.7](#).

Type	Meaning
<code>day</code>	Day of a month
<code>month</code>	Month of a year
<code>year</code>	Year of the Gregorian calendar
<code>weekday</code>	Weekday in the Gregorian calendar
<code>weekday_indexed</code>	The n -th weekday of a month
<code>weekday_last</code>	Last weekday of a month
<code>month_day</code>	Specific day in a specific month
<code>month_day_last</code>	Last day in a month
<code>month_weekday</code>	The n -th weekday in a month
<code>month_weekday_last</code>	Last weekday in a month
<code>year_month</code>	Specific month in a year
<code>year_month_day</code>	Specific day, month, and year
<code>year_month_day_last</code>	Last day in a specific month and year
<code>year_month_weekday</code>	The n -th weekday in a month and year
<code>year_month_weekday_last</code>	Last weekday in a month and year

Table 28.7 The complete list of calendar types in `chrono::` from C++20.

But What Is a Calendar?

A *calendar* is a collection of dates, each of which has a specific name. In the `<chrono>` sense, `sys_date` is the canonical calendar. A calendar that can be converted to and from `sys_date` can be converted to any other calendar.

The standard currently supports only the *Gregorian calendar* and `sys_days`. However, other calendars can be easily added. You might need an ISO week-numbering calendar, the Julian calendar, or the Islamic calendar, for example. In the C++ standard, the Gregorian calendar is referred to as the *civil calendar* as it is the most widely used calendar.

You will find some type aliases, suffixes, and constants that refer to the civil calendar. The constant `November` is part of the civil calendar, although of course there is also a November in other calendars.

The class for the civil calendar is `year_month_day`, with the components `year`, `month`, and `day`. Note that `year` here does not denote a duration, but rather the “name” of a year in the calendar sense. However, you can subtract two `year` instances from each other to get a duration.

It is important that an `int` is not implicitly converted to a year. This prevents careless mistakes. A year literal is marked with `y`, so `2024y`:

```
// https://godbolt.org/z/zMqjqMeav
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
    year y{2021y};
    std::cout << y << "\n";
    month m{October};
    auto result = m + months{3}; // 'months', not 'month': 3 months later
    std::cout << result << "\n"; // overflowed to January
    weekday wd{Thursday};
    auto result = wd + days{4}; // 'days', not 'day': 4 days later
    std::cout << result << "\n"; // overflowed to Monday
    weekday sun1{0}; // 0 is Sunday
    weekday sun2{7}; // 7 is also Sunday
    std::cout << sun1 << "\t" << sun2 << "\n";
    weekday_indexed wdi{wd, 4}; // unspecified 4th Thursday
    std::cout << wdi << "\n"; // Output: Thu[4] — as given by chrono
                                // unspecified values
}
```

And when you put the pieces together, you get this:

```
// https://godbolt.org/z/6z48xvvvM
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
    year this_year{2021y};
    year last_year{2020y};
```

```

year_month_day ymd{this_year, October, day{28}};
std::cout << ymd << "\n";
month_weekday mwd{November, Thursday[4]}; // in an unspecified year
std::cout << mwd << "\n";                // Output: Nov/Thu[4]
month_day_last mdlast{February};          // last day of an unspecified February
year_month_day_last leap{last_year, mdlast}; // Add year
year_month_day_last noleap{this_year, mdlast}; // Add year
std::cout << leap << "\t" << leap.day() << "\n"; // Output: 2020-02-29 29
std::cout << noleap << "\t" << noleap.day() << "\n"; // Output: 2021-02-28 28
}

```

You can create a date using the `year_month_day` constructor. To prevent dangerous mix-up errors, do not simply pass three integers to the constructor. For example, `year_month_day{10, 11, 12}` should not work: What is the day, what is the month, and what is the year? Instead, you create instances of the types `year`, `month`, and `day`, which you then pass to the constructor:

```
year_month_day ymd{year{2024}, month{11}, day{14}};
```

It looks nicer with the predefined suffixes like `"y`, constants like `November`, and the `/` operator, which combines the individual elements into a date:

```

// https://godbolt.org/z/4s69W7qd3
auto a_date = 2024y/11/14d;
auto b_date = 2024y/11/14;
auto c_date = 14d/11/2024;
auto d_date = November/14/2024;

```

The following orders are possible: `month/day/year`, `day/month/year`, and `year/month/day`. One of the first two elements must be a calendar type—so, for example, not `14/10/2024y`, because `14/10` would be recognized as integer division. The `/` operator cleverly creates “incomplete” types as intermediate values until you reach the complete date.

You can also generate the last day of a month:

```
auto ymd = last/February/2024; // last day in February 2024: 2024-02-29
```

Isn't that useful? This also makes the somewhat strange-looking “last” and “unspecified” types from [Table 28.7](#) make sense. These are generated as intermediate results when you create a date with the `/` operator. However, they are also useful in further operations, as you will see, because you can even perform calculations with these intermediate results.

```

// https://godbolt.org/z/Md38b9zvf
#include <chrono>
#include <iostream>
#include <format>

```



```
using namespace std::chrono;
int main() {
    auto ymd = last/February/2024;    // last day in February 2024: 2024-02-29
    std::cout << ymd << "\n";        // the output with << is simple
    std::cout << std::format("{:%Y-%m-%d}\n", ymd); // format is more flexible
    std::cout << std::format("{:%e. %B %y}\n", ymd); // much more flexible!
}
```

Listing 28.28 Output of date and time with format.

The stream outputs 2024/Feb/last—which is correct, but not helpful. With format, you can output both 2024-02-29 and 29. February 24, and so on.

To parse strings into a data type from `<chrono>`, since C++20 there are `chrono::from_stream` and `chrono::parse`.

Invalid Dates and Error Handling

The calendars from `<chrono>` leave it up to you how to handle invalid dates. For example, you can create November 31, which is an invalid date. It is neither automatically clamped to November 30 nor rolled over into December, nor is an exception thrown. It is up to you how to handle the invalid intermediate result:

```
auto ymd = 31d/October/2024;
ymd += months{1};
```

ymd is now 2024y/November/31d. If you want this to automatically be limited to November 30, then use `last`:

```
if (!ymd.ok())
    ymd = ymd.year()/ymd.month()/last;
```

If you want to round into December, convert to `sys_days`. The data type ensures the wraparound at the month's boundary:

```
if (!ymd.ok())
    ymd = sys_days{ymd};
```

Here you can see how to use `last` and the classes to handle invalid dates.

So you can generate a specific day of a month or the last day of a month. But what about, for example, the second Tuesday of a month—the day you always expect Windows updates? Use the `[]` notation for that.⁶

```
// https://godbolt.org/z/19vf9jfTM
#include <chrono>
#include <iostream>
```

⁶ C++20's *chrono* Calendars and Time Zones in MSVC, Miya Natsuhara, <https://www.youtube.com/watch?v=Dq7rqjatxz8>, 2022-03-08, [2023-09-19]

```
using namespace std::chrono;
int main() {
    for (int mo = 1; mo <= 12; ++mo) {
        year_month_weekday patch_tues{mo/Tuesday[2]/2024y};
        year_month_day ymd{sys_days{patch_tues}};
        std::cout << ymd.month() << " " << ymd.day() << "\n";
    }
}
```

Listing 28.29 Always the second Tuesday of a month.

Note how `Tuesday[2]` refers to the second Tuesday of the month. First, we create a `year_month_weekday` object, then we convert it to the universal `sys_days` calendar, and finally to a `year_month_day` calendar for output.

Time Zones

Since C++20, the standard library can also handle time zones. This is a very complex topic that we can only touch on here. The most important types involved are as follows:

- `time_zone`
- `zoned_time`
- `tzdb`
- `tzdb_list`
- `time_zone_link`
- `ambiguous_local_time`
- `nonexistent_local_time`

For a time zone and a time within it, there are the `time_zone` and `zoned_time` data types:

- **`time_zone`**
Represents the time zone of a specific geographic region. It consists of the name, the UTC offset, any daylight saving time rules, and so on—for example, `Europe/Berlin` or `America/New York`.
- **`zoned_time`**
Combines `time_zone` and `time_point` to represent a point in time in a specific time zone. Provides `local_time` or a time zone–less `sys_time`.

To convert, use the constructor of `zoned_time` with a `local_time`:

```
auto autumn = local_days{2d/June/2024} + 16h + 33min;
std::cout << zoned_time{"Europe/Berlin", autumn} << '\n';
```

Conversions to and from `sys_time` and `local_time` are usually straightforward. However, there are some tricky edge cases that always cause trouble. And what is to blame? Daylight saving time.

There are local times that do not exist at all, and some that exist twice. In Europe/Berlin, for example, the time 02:01:00 on March 31, 2024, does not exist. Instead, on October 27, 2024, you have the opportunity to interpret the time 02:01 as daylight saving time and thus as 00:01 UTC, or as standard time and thus as 01:01 UTC.

The C++ standard allows you to handle these cases. When you convert `local_time` to `sys_time`, two types of exceptions can be thrown:

- **`nonexistent_local_time`**
The local time does not exist.
- **`ambiguous_local_time`**
The local time exists twice. You can decide with `choose::earliest` or `choose::latest` which one you mean.

```
// https://godbolt.org/z/ezn568nof
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
    auto spring = local_days{31d/March/2024} + 2h + 1min;
    try {
        auto zt = zoned_time{"Europe/Berlin", spring};
    } catch (const nonexistent_local_time& e) {
        std::cout << e.what() << '\n'; // Exception thrown: does not exist
    }
    auto autumn = local_days{27d/October/2024} + 2h + 1min;
    try {
        auto zt = zoned_time{"Europe/Berlin", autumn};
    } catch (const ambiguous_local_time& e) {
        std::cout << e.what() << '\n'; // Exception thrown: already exists
    }
    std::cout << zoned_time{"Europe/Berlin", autumn, choose::earliest} << '\n';
    // Output: 2024-10-27 02:01:00 CEST
    std::cout << zoned_time{"Europe/Berlin", autumn, choose::latest} << '\n';
    // Output: 2024-10-27 02:01:00 CET
}
```

Listing 28.30 Conversion to a time with time zone and the exceptions.

The last two conversions went smoothly, while the first two threw an exception. Their outputs describe the errors quite thoroughly:

```
2024-03-31 02:01:00 is in a gap between
2024-03-31 02:00:00 CET and
2024-03-31 03:00:00 CEST which are both equivalent to
2024-03-31 01:00:00 UTC
```

2024-10-27 02:01:00 is ambiguous. It could be
 2024-10-27 02:01:00 CEST == 2024-10-27 00:01:00 UTC or
 2024-10-27 02:01:00 CET == 2024-10-27 01:01:00 UTC

If you are dealing with multiple time zones, the *time zone database* comes into play:

■ **tzdb**

This type stores the data of the time zone database. You get it with the `get_tzdb()` function. There are the `zones`, `version`, and `links` vectors as well as the `version` field. With the `current_zone()` and `locate_zone(string_view tz_name)` methods, you get a `const time_zone*`.

■ **tzdb_list**

You can use the `get_tzdb_list()` method to get a list of all time zones, and multiple versions may also be stored.

■ **time_zone_link**

This represents an alternative designation of a time zone.

```
// https://godbolt.org/z/YTKK3vT7s
#include <chrono>
#include <iostream>
#include <ranges> // views::transform, views::filter
#include <algorithm> // ranges::for_each
using namespace std; namespace c = std::chrono;
namespace v = std::views; namespace r = std::ranges;
int main() {
    auto show_name = [](const string_view name) { cout << name << ' '; };
    const auto& db = c::get_tzdb();
    auto names = db.zones
        | v::transform([](const c::time_zone& z) {return z.name();})
        | v::filter([](const string_view name) {
            return name.starts_with("Europe/Be");});
    r::for_each(names, show_name);
    cout << " <- Europe/Be*\n"; // Output: Europe/Belgrade Europe/Berlin
    r::for_each(
        db.links
        | v::filter([](const c::time_zone_link& l){
            return l.target()=="Europe/Berlin";})
        | v::transform([](const c::time_zone_link& l)->string_view {
            return l.name();})
        , show_name);
    cout << " <- Links to Europe/Berlin\n"; // Output: Arctic/Longyearbyen ...
}
```

Listing 28.31 The time zone database.

Standards and Implementations

The standard library defines that the IANA time zone database should be used as the basis for implementations. However, this is not always easy because the database is not available on every system, and furthermore, it is difficult to keep it up to date. Linux systems usually have the database installed (check in `/usr/share/zoneinfo`), but Windows systems do not. However, the Microsoft C++ compiler comes with the ICU library, which contains a similar database. The differences are small but present.

Compile-Time Fractions: “ratio”

In [Table 28.6](#), you saw that the second template parameter of `duration` is of type `ratio`. Indeed, this `ratio` handles all the conversion that is so useful in `<chrono>`. And what is useful for time units can also be helpful elsewhere.

`std::ratio` is a fraction that is evaluated at compile-time. `ratio<1,1000>` is the compile-time equivalent of $\frac{1}{1000}$. You can perform calculations with different `ratio` instances. The compiler even performs simplifications. This means that if you multiply `ratio<1000,1>` by `ratio<1,1000>`, the result is not `ratio<1000,1000>` but `ratio<1,1>`. This is important so that the compiler knows exactly when types are identical and when it needs to convert. If it is a “unit fraction,” then you can omit the second template parameter because its default is one. Thus, `ratio<88>` is the same as `ratio<88,1>`.

“ratios” Are Not Values, but Types!

To emphasize once again: the `ratio<1,2> a{}` and `ratio<1,3> b{}` variables have different types. For example, if you write a `void f(ratio<1,2> v)` function, you cannot call it with `f(ratio<1,3>{})`. It makes little sense to create variables of `ratio`, as `a+b` does not support `ratio` either. However, `ratio` is part of the `duration<ratio<...>>` type, for example. There it serves to ensure that when you add second `a` to minute `b`, you get an automatic conversion.

So that you don't always have to write `ratio<1,1000>` and the like, the standard library predefines the commonly occurring fractions shown in [Table 28.8](#). The ones with question marks `?` are available if `intmax_t` can represent them. *quecto*, *ronto*, *ronna*, and *quetta* are part of the standard only from C++26 onward.

You can use `ratio<>::num` and `ratio<>::den` to determine the numerator and denominator respectively of any fraction.

You can also easily add your own fractions. And from then on, you don't need to worry about conversions anymore. Similar to how `<chrono>` demonstrates, you get operators and conversions delivered to you for free:

```
// https://godbolt.org/z/ThM8q1zEP
#include <iostream>
#include <ratio>
using std::cout; using std::endl;
int main() {
    using oneThird = std::ratio<1,3>;
    using twoFourths = std::ratio<2,4>;
    cout << oneThird::num << "/" << oneThird::den << endl;    // Output: 1/3
    cout << twoFourths::num << "/" << twoFourths::den << endl; // Output: 1/2
    using sum = std::ratio_add<oneThird,twoFourth>;           // add
    cout << sum::num << "/" << sum::den;                        // Output: 5/6
}
```

Typedef	Ratio	Value	Typedef	Ratio	Value
<i>quecto</i>	<code>ratio<1,10...30...00></code>	10^{-30} ?	<i>deca</i>	<code>ratio<10,1></code>	10^1
<i>ronto</i>	<code>ratio<1,10...27...00></code>	10^{-27} ?	<i>hecto</i>	<code>ratio<100,1></code>	10^2
<i>yocto</i>	<code>ratio<1,10...24...00></code>	10^{-24} ?	<i>kilo</i>	<code>ratio<1000,1></code>	10^3
<i>zepto</i>	<code>ratio<1,10...21...00></code>	10^{-21} ?	<i>mega</i>	<code>ratio<1000000,1></code>	10^6
<i>atto</i>	<code>ratio<1,10...18...00></code>	10^{-18}	<i>giga</i>	<code>ratio<1000000000,1></code>	10^9
<i>femto</i>	<code>ratio<1,10...15...00></code>	10^{-15}	<i>tera</i>	<code>ratio<10...12...00,1></code>	10^{12}
<i>pico</i>	<code>ratio<1,10...12...00></code>	10^{-12}	<i>peta</i>	<code>ratio<10...15...00,1></code>	10^{15}
<i>nano</i>	<code>ratio<1,1000000000></code>	10^{-9}	<i>exa</i>	<code>ratio<10...18...00,1></code>	10^{18}
<i>micro</i>	<code>ratio<1,1000000></code>	10^{-6}	<i>zetta</i>	<code>ratio<10...21...00,1></code>	10^{21} ?
<i>milli</i>	<code>ratio<1,1000></code>	10^{-3}	<i>yotta</i>	<code>ratio<10...24...00,1></code>	10^{24} ?
<i>centi</i>	<code>ratio<1,100></code>	10^{-2}	<i>ronna</i>	<code>ratio<10...27...00,1></code>	10^{27} ?
<i>deci</i>	<code>ratio<1,10></code>	10^{-1}	<i>quetta</i>	<code>ratio<10...30...00,1></code>	10^{30} ?

Table 28.8 All predefined “ratio” typedefs. The subscripted digits between “...”s indicate the total number of zeros in the omissions.

Even at the definition stage, the compiler simplifies `ratio<2,4>` to `ratio<1,2>`. When adding $\frac{1}{3}$ and $\frac{1}{2}$, the fractions must first be expanded to sixths to obtain $\frac{5}{6}$ —that is, `ratio<5,6>`.

Besides `ratio_add`, there are more options for arithmetic computation and comparison at compile time, which I have listed in [Table 28.9](#).

Operator	Description
ratio_add	Add
ratio_subtract	Subtract
ratio_multiply	Multiply
ratio_divide	Divide
ratio_equal	Comparison for equality
ratio_not_equal	Comparison for inequality
ratio_less	Comparison for less than
ratio_less_equal	Comparison for less than or equal to
ratio_greater	Comparison for greater
ratio_greater_equal	Comparison for greater or equal

Table 28.9 Arithmetic and comparisons for “ratio” at compile time.

Otherwise, `<chrono>` contains the best examples of a meaningful use of `ratio`.

For example, it would have been possible in [Listing 28.27](#) instead of

```
using ten_days = std::chrono::duration<int, std::ratio<86400*10>>
```

to write the following:

```
using ten_days = std::chrono::duration<int,
    std::ratio_multiply<std::ratio<10>,           // 10 * ...
    std::ratio_multiply<std::ratio<24>, hours::period> // ... 24 hours
    >
>;
```

Here, `10*24*hours` results in a ten-day span. Each duration type has period defined as using. It’s defined as a ratio of what you requested with the template. For example for hours the definition `hours::period` will be `ratio<3600>`. That sounds more complicated than it looks in code (slightly simplified):

```
template<typename Rep, typename Period>
class duration {
    using rep = Rep;
    using period = Period;
    // ...
};
using hours = duration<int, ratio<3600>>;
```

Time Points: “time_point”

While `duration` represents some relative duration, `time_point` is absolute. For `<chrono>`, a `time_point` is a firmly connected pair of a duration and a reference point.

For example, the reference point could be January 1, 2000, and the duration 6,518 days; then the pair of these two values would name the time point as November 5, 2017.

Specifically in C++, for example, `10000s` is 10,000 seconds, while

```
time_point<system_clock, seconds> tp{10000s};
```

is 10,000 seconds after 1/1/1970, which is 1970-01-01 02:46:40 UTC (that January 1, 1970, is the start of `system_clock` is not a must, so be careful here).

Since C++20, there are aliases for `time_point<system_clock, seconds>` such as `sys_time<seconds>` and `sys_seconds`. The `sys_days` alias has also been added.

However, `time_point` and `duration` work well together. You can take the difference between two time points and get a duration—as in C++:

```
// https://godbolt.org/z/q1znGWb85
#include <iostream>
#include <chrono>
int main() {
    using namespace std::chrono;
    time_point<system_clock, seconds> t1{10000s};
    time_point<system_clock, seconds> t2{50000s};
    auto dur = t2 - t1;
    std::cout << duration_cast<hours>(dur).count() << "h"; // Output: 11h
}
```

The compiler does not allow you to *add* two time points, as that would not make sense. However, you can add duration to a `time_point` and so on.

Every `time_point` is bound to a time unit—in this example, seconds. As with `duration`, the compiler allows implicit conversions to more precise `time_point` types but prevents those to less precise ones. With a `time_point_cast`, you can always force a conversion.

To convert a `time_point` to a duration, use the `time_since_epoch()` method.

Clocks

A clock is a triplet of

- a duration in the form of a `duration` type,
- a point in time in the form of a `time_point` type, and
- a function to get the current time.

Here is a somewhat simplified representation of any clock:

```
struct some_clock {  
    using duration = microseconds;           // Granularity of the clock  
    using time_point = time_point<some_clock>; // Return type of now()  
    static time_point now() noexcept;        // get current time  
};
```

I have omitted the `rep`, `period`, and `is_steady` members for better clarity. `microseconds` is mentioned only as an example; each clock defines its own granularity here. The central function is `now()`, which I will explain to you in more detail shortly.

The likelihood that you will need to define new clocks yourself is very low. The standard library comes with a few clocks (up to C++20, only `system_clock`, `steady_clock`, and `high_resolution_clock`), which you will use depending on the application:

■ **`std::chrono::system_clock`**

Use this clock if you need a reference to a calendar. With this clock, you can get the current time or date. It provides `sys_time` and handles leap seconds as we do in everyday life. When you format the seconds, you expect the usual "00" to "59". This clock can store values from midnight, January 1, 1970 (UTC). However, this is only guaranteed from C++20 onward; before that, the starting point was not defined.

■ **`std::chrono::steady_clock`**

This is more of a stopwatch. Time measurements within your program are possible with this. It has no relation to a time of day or a calendar.

■ **`std::chrono::high_resolution_clock`**

This is usually just an alias for one of the other two clocks, depending on which system the clock is more finely granulated for time measurement. Implementations can provide a particularly accurate clock here.

■ **`std::chrono::file_clock`**

This is the same clock as `std::filesystem::file_time_type::clock` and is returned by functions like `filesystem::last_write_time()`. The starting point of the clock is unspecified. However, you can use `clock_cast` to convert to `sys_time`, which has a specified starting point.

■ **`std::chrono::utc_clock`**

Delivers `utc_time`, which is like `sys_time` but accounts for leap seconds. When you calculate differences across the boundaries of leap seconds, they are accounted for with `utc_time`. When you format `utc_time`, account for a second "60", not just "00" to "59".

■ **`std::chrono::gps_clock`**

The `gps_clock` delivers `gps_time` and starts on June 1, 1980. It does not account for leap seconds. It can be, but does not have to be, very accurate. It is currently 18 seconds ahead of `sys_time` and `utc_time`.

■ `std::chrono::tai_clock`

International Atomic Time (abbreviated to TAI from its French name) is a very accurate time that does not account for leap seconds. It starts on January 1, 1958. When you format `tai_time`, it is always 19 seconds ahead of `gps_time`.

Local Pseudo Clock

The `local_t` clock is a clock for intermediate calculations when you have a time point to which you want to add a time zone later. This clock does not have a `now()` method.

If you want the current time of a clock, such as `steady_clock`, you can get it as follows:

```
steady_clock::time_point tp = steady_clock::now();
```

Or directly with `auto`:

```
auto tp = steady_clock::now();
```

In general, when working with `<chrono>`, you can very often use `auto` and thus do not have to remember all return types.

You use this, for example, to measure the time of a function call.

```
// https://godbolt.org/z/3MGbYn4d5
#include <iostream>
#include <chrono>
long fib(long n) { return n<2L ? 1L : fib(n-1L)+fib(n-2L); }
int main() {
    using namespace std::chrono;
    auto t0 = steady_clock::now(); // On your marks, get set ...
    auto res = fib(17);           // ... go!
    auto t1 = steady_clock::now(); // Stop!
    std::cout << "Result: " << res << "\n"; // Output: Result: 2584
    std::cout << "Time: " << nanoseconds{t1-t0}.count() << "ns\n";
    // Output: Time: 50727ns (e.g.)
}
```

Listing 28.32 Simple timing of a function call.

Simply measure the time before and after the call with `now()` and get a `time_point` each time. Between these two, you calculate the difference with `t1-t0` and get a duration. For the output, you then convert this duration into a time unit appropriate for the measurement—here, `nanoseconds{...}`. And you're done.

Converting Times and Leap Seconds

Since C++20, you can use `clock_cast<>` to convert values between different `time_point<>` types. The starting points of the clocks and the leap seconds are taken into account.

For this, there is the `clock_time_conversions<>` trait. Clocks that support `to_utc`, `from_utc`, `to_sys`, and `from_sys` can then be converted using `clock_cast<>`.

```
// https://godbolt.org/z/bsqP75Kr8
#include <iostream>
#include <chrono>
using namespace std::chrono;
int main() {
    tai_time tai_now = tai_clock::now();
    std::cout << tai_now << "\n"; // Output: 2023-10-19 01:13:06.2810034
    utc_time utc_now = clock_cast<utc_clock>(tai_now);
    std::cout << utc_now << "\n"; // Output: 2023-10-19 01:12:29.2810034
}
```

Listing 28.33 Converting time points with “`clock_cast`”.

At the inception of these clocks, they were already 10 seconds apart. The additional 27 leap seconds since then lead to a discrepancy of 37 seconds, which will continue to change. The `clock_cast` function takes this into account.

The *leap seconds* are determined by an international committee and synchronize Earth's rotation with atomic time. They are usually added or removed on June 30 or December 31. Since establishment in 1972, there have been a total of 27 inserted leap seconds, the last one on December 31, 2016. There has not been a skipped second so far, but it could happen.

If you want a coarser time unit that you cannot implicitly convert due to loss of precision, then you have two already explained options:

```
// https://godbolt.org/z/rz38hc1cx
auto res = fib(45);
// ...
std::cout << "Time: " << duration_cast<seconds>(t1-t0).count() << "s\n";
// Output: Time: 7s (e.g.)
std::cout << "Time: " << duration<double>(t1-t0).count() << "s\n";
// Output: Time: 7.35303s (e.g.)
}
```

With `duration_cast<>`, you can accept loss of precision in the output. If you convert to a floating-point duration, the compiler allows the loss of precision—which with floating-point numbers is not actually a loss.

If you don't need a stopwatch but a wall clock, then use `system_clock`. The point in time it describes results from the elapsed duration since the epoch. *Epoch* is the implicit start of this clock. You get the duration since the epoch with `time_since_epoch()`:

```
// https://godbolt.org/z/1xbhhTdEv
#include <iostream>
#include <chrono>
int main() {
    using namespace std::chrono;
    auto tp = time_point_cast<seconds>(system_clock::now());
    auto d = tp.time_since_epoch();
    std::cout << d.count() << "s\n";
    std::cout << duration<double, std::ratio<86400>>(d).count() << "days\n";
}
```

On all systems known to me, the epoch corresponds to 1970-01-01 00:00:00.⁷ `time_since_epoch()` returns a duration here, usually based on seconds, which you can manipulate as usual.

The output is currently as follows for me:

```
1488837296s
17231.9days
```

28.4.2 Predefined Suffixes for User-Defined Literals

In the standard library, three types of user-defined literals are defined—*user-defined* in the sense that they are defined using `operator""`. This means that you can write literals of the respective type directly in the source code using the defined suffixes.

Custom User-Defined Literals

Remember that you are only allowed to write custom user-defined literals that begin with an underscore `_`. All others are reserved for the present and future of the standard library.

I have already mentioned the literals during the discussion of the respective types. However, I want to list them all together here and mention the commonalities.

The predefined user-defined literal suffixes are in [Table 28.10](#). Each header defines its literals in its own inline namespace. Also, `std::literals` is an inline namespace:

⁷ Microsoft plans to set this point to 1.1.1601, which aligns better with other Windows APIs. So be careful with this assumption.

- `<string>` defines inline namespace `std::literals::string_literals`
- `<complex>` defines inline namespace `std::literals::complex_literals`
- `<chrono>` defines inline namespace `std::literals::chrono_literals`

Because the literals are in their own namespaces, you can fetch them separately. But because they are inline namespaces, you can fetch them all with a single using:

- using namespace `std` makes the suffixes for `string` and `complex` available along with other identifiers of the standard library if you have included the respective headers. `chrono` is in its own namespace.
- using namespace `std::chrono` makes the suffixes from `<chrono>` available along with other identifiers when you have included `<chrono>`.
- using namespace `std::literals` also makes all suffixes available, but other identifiers must still be addressed with `std::`.
- using namespace `std::literals::chrono_literals` makes only the literals defined in `<chrono>` available.
- using `std::literals::string_literals::operator""s` makes only the suffix `""s` available for `string`, but not the other suffixes, especially not `""s` for second from `<chrono>`. This also works for all other `operator""s`.

Suffix	Type	Header
<code>""s</code>	<code>std::string</code>	<code><string></code>
<code>""sv</code>	<code>std::string_view</code>	<code><string_view></code>
<code>""i</code>	<code>std::complex<double></code>	<code><complex></code>
<code>""if</code>	<code>std::complex<float></code>	
<code>""il</code>	<code>std::complex<long double></code>	
<code>""h</code>	<code>std::chrono::hours</code>	<code><chrono></code>
<code>""min</code>	<code>std::chrono::minutes</code>	
<code>""s</code>	<code>std::chrono::seconds</code>	
<code>""ms</code>	<code>std::chrono::milliseconds</code>	
<code>""us</code>	<code>std::chrono::microseconds</code>	
<code>""ns</code>	<code>std::chrono::nanoseconds</code>	
<code>""d</code>	<code>std::chrono::day</code> (since C++20)	
<code>""y</code>	<code>std::chrono::year</code> (since C++20)	

Table 28.10 The predefined literals in the standard library for `operator""` suffixes.

What you use depends on the situation. In particular, `"s` for string from `<string>` and `"s` for seconds from `<chrono>` can be cleanly separated this way if needed.

You can find plenty of examples, such as in [Chapter 4, Listing 4.17](#) for `<string>`, [Chapter 4, Listing 4.39](#) for `<complex>`, and [Listing 28.24](#) for `<chrono>`. Here is a summary with several different literals:

```
// https://godbolt.org/z/o359TThMj
#include <iostream>
#include <string>
#include <chrono>
#include <complex>

using std::cout;

int main() {
    { using namespace std;
      cout << "string"s << "\n";           // string
      cout << (1.2+3.4i) << "\n";         // complex
    }
    { using namespace std::chrono;
      cout << (35ms).count() << "ms\n";    // chrono
    }
    { using namespace std::literals;
      cout << (41s).count() << "ms\n";     // chrono seconds
      cout << "text"s << "\n";            // string
    }
    { using namespace std::chrono;
      cout << (4h).count() << "h\n";       // chrono hours
    }
    { using namespace std::literals::chrono_literals;
      cout << (16min).count() << "min\n";  // chrono minutes
    }
    { using std::literals::string_literals::operator"s";
      cout << "letters"s << "\n";         // string
    }
}
```

Note that although `"s` exists for string and seconds, you can use both simultaneously because the overloads for `operator"s` do not conflict. Text literals like `"text"s` become string, and numeric literals like `41s` become seconds.

The GCC and “complex” Suffixes

For GCC, the `""i`, `""if`, and `""il` suffixes for complex literals currently only work as expected if you specify `-fno-ext-numeric-literals` during compilation. Definitions from the `<complex.h>` C header along with long-existing GCC extensions otherwise confuse the compiler.

28.5 System Error Handling with “system_error”

Errors caused by the operating system or a similarly low level and reaching the program are typically returned to the caller as a *numeric value*. On a POSIX system, the “Permission denied” error is reported, for example, with the error code 1 (constant `EPERM`).

To write portable code, you need to translate these codes into general—portable—representations. In doing so, you must deal with many possible systems. And just to be able to do that, you also need to be able to access the system-specific codes again. On the one hand, you want a portable translation; on the other hand, you want system-specific low-level access.

The `<system_error>` header tries to bridge this gap. It contains lightweight `error_code` objects that contain system-specific `int` error codes on the one hand, but on the other hand refer to more abstract and portable `error_condition` objects.

These error objects can also be used without the exception class, which developers who work without exceptions in their environment will appreciate. If you use exceptions, you can trigger an `error_code` along with a text message as a `system_exception`.

Extensibility is addressed by the fact that these two fields are each associated with an `error_category`. Here, users can add their own error groups or recognize those defined by the system.

28.5.1 Overview

The `<system_error>` header primarily defines the following components:

- **class `error_code`**
Represents a (*system-*)*specific* error value that can be returned by an operation, such as a system function call.
- **class `error_condition`**
A *generic* error condition against which one checks in their *portable* code.

- **class error_category**
Classification of errors into *groups/families*, also an abstract base class for translation into readable text.
- **class system_error**
Is an exception class that contains an `error_code` when an error is propagated by exception.
- **enum class errc**
Is an enumeration of general-purpose values derived from the POSIX standard of type `error_condition`.
- **is_error_code_enum<>, is_error_condition_enum<>, make_error_code, make_error_condition**
Implement mechanisms to convert `enum class` values into `error_code` or `error_condition` values.
- **generic_category()**
Returns an `error_category` instance that can interpret `errc`-based *codes* and *conditions*.
- **system_category()**
Returns an instance to interpret errors from the operating system.

28.5.2 Principles

Although `system_error` defines its own exception class, throwing an exception is not always the right way to handle an error. Sometimes, the additional return value of a function simply provides information that you want to incorporate into your program's control flow. A broken connection to a network server might be propagated as an exception, but if you are trying multiple servers during connection setup, you should not necessarily implement this in a loop with exceptions. And do not forget that in some areas, exceptions are entirely undesirable: In *real-time systems*, the requirements for deterministic timing behavior are so high that implementers sometimes avoid exceptions altogether. And in *embedded systems*, the additional memory consumption that some exception implementations bring cannot be neglected. Therefore, C++ pragmatically supports both types of error handling—return values with error codes as well as exceptions—and bridges the two worlds.

In case of an error, the `errno` variable often stores more detailed information about the nature of the error. Many functions from `<stdio>` and `<math>` store an error code here in case of failure. On POSIX-compliant platforms, many system functions provide details in `errno` when a function call fails. On Windows, `GetLastError` is often used additionally or instead. And the more libraries you use, the more sources from which error codes can be determined are added. C++ takes into account the possibility that error codes from different sources need to be evaluated and supports the extension of these by libraries. When that happens, the interface of existing functions must not change.

When users and third-party providers are given the means to extend the error code system, both the low-level error codes (which may vary from system to system) need to be mapped somewhere, and users need to be offered a portable interpretation. For example, someone implementing an HTTP server wants to pass on the error codes defined in the RFC standard in a way that everyone can understand, but must rely on operating system-specific error codes to do so. We will demonstrate this with an example in the next section. Originally, it was intended that libraries (the standard library, but also those from third-party providers) would only offer users the already interpreted error codes. However, the original information is also important for problem analysis, and therefore it must somehow be present in *bug reports* or *logging*. Therefore, error code handling must offer both: low-level error codes and interpreted/portable values.

28.5.3 “error_code” and “error_condition”

The `error_code` and `error_condition` classes look almost identical, but they have different roles. `error_code` transports system-dependent errors, while `error_condition` is used for a more portable classification.

So it depends on what you want to achieve. Consider the following example.

```
// https://godbolt.org/z/1xnY8vd8Y
#include <system_error> // error_code
#include <string>
void create_dir(const std::string& pathname, std::error_code& ec);
void run() {
    std::error_code ec;
    create_dir("/some/path", ec);
    if(!ec) { // Success ...
    } else { // Failure ...
    }
}
```

Listing 28.34 Very simple example of how to check the success of an operation.

Here we are only interested in success or failure, which is why `error_code` can be easily converted into a `bool` using `if(!ec)`. This behavior is derived from checking the return value of a function to see if an error occurred—unfortunately, this varies with each function, such as `FILE *f=fopen("f.txt", "r"); if(f==NULL)....`

You can also use the `error_code` to get more information in case of an error: Do you lack permission, does the filename already exist, is the pathname too long, does the parent directory not exist? And so on. In any case, `ec` contains a system-specific value for this—similar to the `errno` variable in traditional code (but only in case of an error).

Suppose you were more interested in whether a file or directory with the name already exists. Then you could determine this with `if(ec.value() == EEXIST)`—but that would

be the wrong way! It would work on a POSIX platform, but on Windows, you would have to compare `if(ec.value() == ERROR_ALREADY_EXISTS)`.

As a rule of thumb: *If you call `value()`, you are probably doing something wrong.*

But how do you instead test the system-specific error code (`EEXIST` or `ERROR_ALREADY_EXISTS`) against the “already exists” event? This is done using the `error_condition`.

You can directly compare the returned `error_code` `ec` using `==` or `!=` with instances of `error_condition`. In this case, the comparison is not for exact identity but for *equivalence*—meaning that it “means the same.”

Therefore, in C++, you must compare the `error_code` `ec` with an `error_condition` object that represents “already exists”—namely, `errc::file_exists`. For the moment, consider `errc::file_exists` (and all other elements of the enum class `errc`) as placeholders for a constant of type `error_condition`. The exact mechanism will become clear as we proceed.

```
// https://godbolt.org/z/TPWc5nexq
#include <system_error> // error_code, errc
#include <string>
void create_dir(const std::string& pathname, std::error_code& ec);
void run() {
    std::error_code ec;
    create_dir("/some/path", ec);
    if(ec == std::errc::file_exists) { // specifically ...
    } else if(!ec) {                  // Success ...
    } else {                          // Failure ...
    }
}
```

Listing 28.35 Comparison of “error_code” with “error_condition”.

To enable the comparison, the `operator==` free functions are overloaded.

```
bool operator==(
    (const error_code& lhs, const error_code& rhs) noexcept;
...(const error_code& lhs, const error_condition& rhs) noexcept;
...(const error_condition& lhs, const error_code& rhs) noexcept;
...(const error_condition& lhs, const error_condition& rhs) noexcept;
```

Listing 28.36 Overloads of “operator==()” (abbreviated).

These indirectly ensure that both `error_code` `EEXIST` and `ERROR_ALREADY_EXISTS` can be compared as equivalent to `error_condition` `errc::file_exists`.

The implementation is responsible for mapping system-specific `error_codes` to general `error_condition` error categories.

Enumerators as Constants for Classes

The *enum class* `errc` is defined as in the following listing.

```
enum class errc {  
    address_family_not_supported,  
    address_in_use,  
    // ...  
    value_too_large,  
    wrong_protocol_type,  
};
```

Listing 28.37 Definition of “enum class `errc`”.

It seems that an *implicit conversion* of the `errc` element to an `error_condition` instance occurs when we then apply the comparison `ec == errc::file_exists`. Almost, but not quite: for the entire system to be extensible, the `error_category` of `ec` must somehow be taken into account. Because there are overlaps in error values, 66 can mean one thing in file operations but something entirely different in memory operations. This is resolved through the `error_category` mechanism. In addition, `errc` here represents a set of constants for `error_condition`. But the library also supports the reverse way—that is, comparing an `enum` element representing an `error_code` with an `error_condition` instance.

Therefore, two steps are necessary for the equivalence check:

1. Decide whether `errc` represents an `enum` for specific error codes or generic error conditions.
2. Associate the value with an `error_category`.

For step 1, the `is_error_code_enum` and `is_error_condition_enum` *type traits* are used. By default, `is_error_code_enum<E>::value` and `is_error_condition_enum<E>::value` yield the value `false`. The standard library now specializes `is_error_condition_enum` and thus defines `is_error_condition_enum<errc>::value == true`.

```
template <>  
struct is_error_condition_enum<errc> : true_type {};
```

Listing 28.38 Specialization of “`is_error_condition_enum`” for “`errc`”.

The implicit conversion can then take place through the templatization of the constructor of `error_condition`—that is, through `template <class E> error_condition(E e)`.⁸ When we then write `ec == errc::file_exists`, the compiler chooses the `bool` operator `==` (`const error_code& a, const error_condition& b`) overload, because the constructor

⁸ `E` stands for the `enum` class, and the overload can only be found if `is_error_condition_enum<errc>::value` is `true`. The exact implementation is left to the library, but it is possible with standard template metaprogramming methods.

provides a way to convert the enum into an `error_condition`. This makes it easy for us to test a given `error_code` against a specific `error_condition`—even if it is only represented by an enum element.

Paths to the Error Category

Now follows step 2. The appropriate `error_condition` was created by the conversion with a constructor. The free function `make_error_condition()` is called in it. This is also overloaded with `errc`:

```
error_condition make_error_condition(errc c) noexcept {
    return error_condition(
        static_cast<int>(e),
        generic_category());
}
error_condition make_error_condition(io_errc c) noexcept /*...*/
error_condition make_error_condition(future_errc c) noexcept /*...*/
```

Listing 28.39 Among the overloads of “`make_error_condition`” is also one with “`errc`”.

The other two are `io_errc` (see `ios_base::failure`) and `future_errc` (see `future_error`).

Here the connection with the corresponding `error_category` is established. `generic_category()` returns a reference to such an instance and could, for example, be implemented as a *singleton*.

For end users, the `make_error_code()` function family can also be interesting—for example, if you are writing portable code and need to use system-specific error codes.

```
// https://godbolt.org/z/n4v78WWod
#include <system_error>
#include <string>

void create_dir(const std::string& pathname, std::error_code& ec) {
    #if defined(_WIN32)
        // Windows implementation, with Windows error codes
    #elif defined(linux)
        // Linux implementation, with Linux error codes
    #else
        // general 'generic' case
        ec = std::make_error_code(std::errc::not_supported);
    #endif
}
```

Listing 28.40 Generating system-specific error codes in a portable program.

28.5.4 Error Categories

There are two main error categories:

- **generic_category**
POSIX-compatible error messages, either used on such a system or on other systems by functions that emulate POSIX functionality, like `fopen` on Windows.
- **system_category**
Room for extensions beyond POSIX or on other systems for a custom range. For example, on Windows, this includes the `GetLastError()` errors.

28.5.5 Custom Error Codes

If you do not want to misuse the standard library error codes in your project, it is recommended to proceed as follows:

- Define an `enum` with your own error codes.
- Then use `make_error_code` in your library with these error codes as parameters.
- Create an overload of `make_error_category` for your error code `enum` and map your error codes to portable `error_category` elements there.
- Specialize the template `std::is_error_condition_enum<>` with the content `true_type` so that the standard library knows that you support `error_category`.

The last step, mapping to `error_condition`, is optional if you do not need the coarse granularity in the consuming code:

```
// https://godbolt.org/z/11Tdr6aG1
#include <system_error>
#include <iostream>
using std::error_code; using std::system_category;
namespace mylib {
    // custom error codes
    enum class errc { LOAD_ERR = 1, UNLOAD_ERR = 2, OTHER_ERR = 3 };
    error_code make_error_code(errc ec) {
        switch(ec) {
            case errc::LOAD_ERR: return error_code((int)ec, system_category());
            case errc::UNLOAD_ERR: return error_code((int)ec, system_category());
            case errc::OTHER_ERR: return error_code((int)ec, system_category());
        }
    }
    error_code run(int arg) {
        if(arg == 667) {
            return make_error_code(errc::OTHER_ERR);
        }
        return error_code{}; // all good.
    }
}
```

```

    }
}
int main() {
    std::error_code ec = mylib::run(667);
    if(!ec) {
        std::cout << "Great, it works!\n";
    } else if (ec == mylib::make_error_code(mylib::errc::OTHER_ERR)) {
        std::cout << "Another error\n";
    } else {
        std::cout << "Nothing happening here\n" << ec;
    }
}

```

Similarly, proceed to support the portable `error_category` as well.

28.5.6 “system_error” Exception

Library functions that prefer to propagate their errors as exceptions use the large aggregate exception class `system_error` from the header of the same name. This class again transports

- an `error_code`, consisting of an `int` detail and an `error_category`; and
- an error message.

This single exception can then be caught and examined in more detail. This is the alternative to many exception classes in a more or less arbitrary hierarchy—which may also differ on different systems and implementations.

For example, library code might look like this, attempting to join a thread:

```

int osSpecificCode = pthread_detach(thread, 0);
if (osSpecificCode != 0)
    throw system_error(
        error_code(osSpecificCode, system_category()),
        "thread::detach failed");

```

You would handle this in user code like this:

```

// https://godbolt.org/z/a8zP84j1x
#include <thread>
#include <iostream>
#include <system_error>
int main() {
    try {
        std::thread().detach(); // this will fail
    } catch(std::system_error& e) {

```

```
std::cout
    << "system_error with code:" << e.code()
    << " message:" << e.what()
    << '\n';
}
```

For me, this outputs the following:

```
system_error with code:generic:22 message:Invalid argument
```

Here, code contains the `osSpecificCode`.

In the standard library, there are already many places where a `system_error` can be thrown. You can handle them like this:

```
// https://godbolt.org/z/cjh93oPcr
#include <iostream>
#include <system_error> // std::make_error_condition, std::ios_errc
int main () {
    // switch to exceptions:
    std::cin.exceptions (std::ios::failbit|std::ios::badbit);
    try {
        std::cin.rdbuf(nullptr); // triggers an exception
    } catch (std::ios::failure& e) { // derived from system_error
        std::cerr << "Error: ";
        if (e.code() == std::make_error_condition(std::ios_errc::stream)) {
            std::cerr << "stream\n";
        } else {
            std::cerr << "other\n";
        }
    }
}
```

So you catch the exception in a catch block. By comparing it with an `error_condition`, you can then check portable error conditions more precisely.

28.6 Runtime Type Information: “<typeinfo>” and “<typeindex>”

The `<typeinfo>` header actually contains only a few things:

- **Operator typeid()**
A function-like operator for determining a unique identification
- **Class type_info**
The return type of `typeid()`

■ Exceptions `bad_cast` and `bad_typeid`

The exceptions that occur in the context of dynamic types

This is a little bit of a lie: `typeid()` is similar to `sizeof()` in that it is not a function but an operator, and `typeid` is even a keyword. However, you *must* include the `<typeinfo>` header before using `typeid()`; otherwise it is an error. Therefore, you can think of `typeid()` as a function defined in the header.

With this “function,” you can get information about a type that you specify as a parameter, which you can use for comparison purposes or, to a limited extent, for debugging.

The type can either be directly the name of a type or an expression. In the latter case, `typeid()` returns information about the type of the expression. However, the expression cannot include calculations like `a+b*c` or even method or function calls.

But what information does `typeid` return? Well, mainly a `const&` to a `type_info` instance; let's call it `ti`. The following things can be done with `ti`:

■ `size_t ti.hash_code()`

Returns a hash value, for example, to quickly test for inequality.

■ `const char* ti.name()`

Returns a textual representation of the type. However, this does not have a standardized format, nor must it be different for all types in a program, nor is it guaranteed that this text will always be the same for a given type. So, it is not very reliable, although informative, and is therefore mainly used for diagnostics.

■ `type_index(ti)`

This important function from the `<typeid>` header is the way to check the equality of two types based on their `type_info` references.

More important than what works is perhaps what *does not* work: You cannot create or copy a `type_info`, and you should not compare them with `==`. It is not guaranteed that two calls `ti1` and `ti2` of `typeid(A)` will return the same `type_info`—in fact, even `ti1 == ti2` is not guaranteed, although `operator==` is overloaded for `type_info`.

If you want to find out whether `ti1` and `ti2` represent the same type, do it like this: `type_index(ti1) == type_index(ti2)`.

`type_index` can also be used as a key for `map` and similar, which is not possible with `type_info`. This allows you to build a mechanism with reliable type names, for example.

```
// https://godbolt.org/z/d1G9ez4KW
```

```
#include <iostream>
#include <typeinfo>
#include <typeid>
#include <map>
#include <string>
```



```
struct Base {
    virtual ~Base() {}
};
struct Derived_One : public Base {};
struct Derived_Two : public Base {};
int main() {
    using std::string; using std::cout; using std::type_index;
    std::map<std::type_index, string> names {
        { type_index(typeid(int)), "int" },
        { type_index(typeid(double)), "double" },
        { type_index(typeid(Base)), "Base" },
        { type_index(typeid(Derived_One)), "Derived_One" },
        { type_index(typeid(Derived_Two)), "Derived_Two" },
        { type_index(typeid(string)), "string" },
        { type_index(typeid(string::const_iterator)), "string" },
    };
    names[type_index(typeid(names))] = "names-map";
    int integer;
    double floating;
    Base base{};
    Base *one = new Derived_One{};
    Base *two = new Derived_Two{};
    // typeid.name() is implementation- and runtime-dependent:
    cout << typeid(integer).name() << '\n';    // On my system: i
    cout << typeid(floating).name() << '\n';    // On my system: d
    cout << typeid(base).name() << '\n';        // On my system: 4Base
    cout << typeid(*one).name() << '\n';        // On my system: 11Derived_One
    cout << typeid(*two).name() << '\n';        // On my system: 11Derived_Two
    cout << typeid(string).name() << '\n';      // For me: Ss
    cout << typeid(string{"World"}.begin()).name() << '\n';
        // For me: N9__gnu_cxx17__normal_iteratorIPcSsEE
    cout << typeid(names).name() << '\n';
        // For me: St3mapISt10type_indexSsSt4lessISO_ESaISt4pairIKSO_SsEEE
    cout << typeid(666/0).name() << '\n'; // Expression not executed! For me: i
    // type_index makes type_infos comparable:
    cout << names[type_index(typeid(integer))] << '\n'; // Output: int
    cout << names[type_index(typeid(floating))] << '\n'; // Output: double
    cout << names[type_index(typeid(base))] << '\n';    // Output: Base
    cout << names[type_index(typeid(*one))] << '\n';    // Output: Derived_One
    cout << names[type_index(typeid(*two))] << '\n';    // Output: Derived_Two
    cout << names[type_index(typeid(string))] << '\n';  // Output: string
    cout << names[type_index(typeid(names))] << '\n';  // Output: names-map
}
```

Listing 28.41 Use “typeid” for reliable type information.

The pure call of `typeid(...).name()` provides a name, but not a reliable one. For me, it returns `int` simply as “i”. And for an iterator or a template like `map`, the output is only limitedly informative.

You counteract this by applying `type_index(...)` to the result of `typeid`—that is, to the `const type_info&`. As you can see in the `names` example, you can also populate a `map` with more readable names this way.

Note that you can use both plain types like `int` or `string::const_iterator` as a parameter for `typeid`, as well as an expression like `integer`, `*one`, or `string{"World"}.begin()`. The expression is not executed; only its type is considered. `typeid(666/0)` would otherwise cause the program to crash.

Finally, compare the results of `typeid(*one)` and `typeid(*two)`. These are polymorphic objects, as both are statically of type `Base`. Dynamically—at runtime—the objects actually have the `Derived_One` and `Derived_Two` types. With `typeinfo`, you can obtain dynamic type information. Note that this is a slightly different use of the `typeid` operator, where it is again necessary for the compiler to execute the expression. Because `Base` contains a virtual method, the compiler recognizes that you need dynamic type information and must therefore execute the expression.

My approach of writing all desired types beforehand into a `map` like `names` is not always feasible in practice. You might not know in advance what types you will encounter. Then you have only two options:

- You have to make do with `type_info::name()`, even if it is not portable.
- You use `demangled_name` from Boost.

The Boost library provides `boost::core::demangled_name(...)`, which returns well-readable type information. Use not `typeid` but the `BOOST_CORE_TYPEID` macro to get the argument for the function. Although Boost cannot guarantee 100% that this name is readable or stable, the library is so well maintained that it is effectively so.

The outputs for templates will never be pretty, but at least they are readable.

```
// https://godbolt.org/z/n54566cqT
#include <iostream>
#include <typeinfo>
#include <string>
#include <map>
#include <boost/core/typeinfo.hpp>
int main() {
    using std::string; using std::cout;
    std::map<int, string> names;
    int i;
    double f;
    // demangled_name
    using boost::core::demangled_name;
```

```
cout<<demangled_name(BOOST_CORE_TYPEID(i))<<'\n';      // Output: int
cout<<demangled_name(BOOST_CORE_TYPEID(f))<<'\n';      // Output: double
cout<<demangled_name(BOOST_CORE_TYPEID(string))<<'\n';  // Output: std::string
cout<<demangled_name(BOOST_CORE_TYPEID(string{}.begin()))<<'\n';
// Output: __gnu_cxx::__normal_iterator<char*, std::string>
cout<<demangled_name(BOOST_CORE_TYPEID(namen))<<'\n';
// Output: std::map<int, std::string, std::less<int>,
//          std::allocator<std::pair<int const, std::string> > >
cout<<demangled_name(BOOST_CORE_TYPEID(666/0))<<'\n';  // Output: int
}
```

Listing 28.42 Boost's “demangled_name” is extremely useful for outputting type names.

28.7 Helper Classes around Functors: “<functional>”

Even more than <numeric>, the functions in <functional> are there to complement other parts of the standard library. You will rarely include <functional> alone. And if you want to solve a “simple” task with <algorithm> or a container, it's worth taking a look at <functional>, where there might already be a solution for it.

If you want to sum up all elements of a container, for example, use `accumulate` from <numeric>. But if you want to use `*` instead of `+`, you can

- either write a lambda `[](auto a, auto b) { return a*b; };`
- or use `multiplies` from <functional>, which does exactly that.

And so it is with many things from <functional>.

I categorize the things offered in <functional> (more or less arbitrarily) into the following areas:

■ Function factories

`function` is the central point for everything callable in C++. `bind` creates a new function object with one less argument from a function object and an argument. `mem_fn` creates a function object from a method. There are several auxiliary constructs in this context.

■ Concrete function objects

For arithmetic (like `multiplies`), comparisons (like `equal_to`), logical (`logical_and`, etc.), and bitwise (`bit_and`, etc.) operations, which you can use instead of a simple lambda.

■ Specialized hash functions

The hash function objects needed for the *unordered containers* are defined here for the built-in types. So, for example, you will find the template specialization `hash<int>` and all its relatives. That should be self-explanatory, so I will not go into further detail here.

28.7.1 Function Objects

In [Chapter 25](#), [Listing 25.15](#), there’s an example of `multiplies` together with `accumulate` from `<numeric>`. The other defined function objects can be found in [Table 28.11](#) to [Table 28.14](#).

Functor	Behavior	Functor	Behavior
<code>plus</code>	$a + b$	<code>divides</code>	a / b
<code>minus</code>	$a - b$	<code>modulus</code>	$a \% b$
<code>multiplies</code>	$a * b$	<code>negate</code>	$-a$

Table 28.11 Function objects of arithmetic operators.

Functor	Behavior	Functor	Behavior
<code>equal_to</code>	$a == b$	<code>less</code>	$a < b$
<code>not_equal_to</code>	$a != b$	<code>greater_equal</code>	$a \geq b$
<code>greater</code>	$a > b$	<code>less_equal</code>	$a \leq b$

Table 28.12 Function objects for comparison operators.

Functor	Behavior	Functor	Behavior
<code>logical_and</code>	$a \&\& b$	<code>logical_not</code>	$!b$
<code>logical_or</code>	$a \ b$		

Table 28.13 Function objects for logical operators.

Functor	Behavior	Functor	Behavior
<code>bit_and</code>	$a \& b$	<code>bit_xor</code>	$a \wedge b$
<code>bit_or</code>	$a b$	<code>bit_not</code>	$\sim b$

Table 28.14 Function objects for bitwise operators.

You can use the function objects to equip containers and algorithms with your own functionality.

For example, in the definition of `std::set`, you will see that `less` from `<functional>` is used here for a comparison operation:

```
namespace std {  
    template<class Key, class Compare=less<Key>, class Allocator=allocator<Key>>  
        class set;  
    // ...
```

So instead of overloading `operator<` for your own data type, you can also specialize the `less` template for your data type, similar to what you would do for `hash` in the context of `unordered_set`:

```
// https://godbolt.org/z/Y3GsPd97d  
#include <set>  
#include <string>  
struct Dragon {  
    std::string name_;  
};  
namespace std {  
    template<> struct less<Dragon> { // Template specialization  
        bool operator()(const Dragon &lhs, const Dragon &rhs) const {  
            return lhs.name_ < rhs.name_;  
        }  
    };  
} }  
int main() {  
    std::set<Dragon> dragons {  
        Dragon{"Smaug"}, Dragon{"Glaurung"},  
        Dragon{"Ancalagon"}, Dragon{"Scatha"}  
    }  
}
```

So if for any reason it is not possible or desired to overload `operator<` for `Dragon`, and you also do not want to specify the template parameter `Compare`, you can achieve the same with a specialization of `less`.

In [Listing 28.43](#), you see a version of the calculator from [Chapter 8, Listing 8.18](#) that uses `<functional>` from start to finish. Instead of a large `switch` statement, this time all possible characters are stored as keys in `maps`. Depending on how the stack should be manipulated, the operations end up in one of four `maps`:

- **binOps**
Remove two elements from the stack, apply the function to them, and store the result back on the stack. Addition is an example of this.
- **unOps**
Remove one element from the stack, apply the function to it, and store the result back on the stack. This calculator does not have such an operation.
- **zeroOps**
Execute a zero-argument function and store the result on the stack. The digits do exactly this.

■ stackOps

Instead of manipulating the stack before and after, the calculator leaves the entire stack to these functions—for example, to clear or print it.

Of course, not everything can be represented with predefined functors, which is why you will see lambdas in use where necessary—and with `val`, even a lambda that returns a lambda. It’s often said that this makes “lambdas as first-class citizens”, which means that they are in most ways values like variables of built-in types or classes and thus can be created, passed as parameters, and even put into containers.

This type of calculator is not generally “better” than using the `switch` statement. However, when it comes to extensibility, this option is the more skillful choice.

There are a few things I would definitely improve “in the real world”:

- Instead of using `std::cout` as a global variable, the operators should be parameterized on `ostream&`.
- The various `find` calls should only be executed when necessary.
- Processing individual digits is illustrative but impractical. Sequences of digits should be whole numbers, with the space as the delimiter.

```
// https://godbolt.org/z/jxvhded75
#include <string>
#include <vector>
#include <iostream>
#include <map>
#include <functional>

std::map<char, std::function<int(int, int)>> binOps { // binary operators
    {'+', std::plus<int>{} },
    {'-', std::minus<int>{} },
    {'*', std::multiplies<int>{} },
    {'/', std::divides<int>{} },
    {'%', std::modulus<int>{} },
};

std::map<char, std::function<int(int)>> unOps { }; // unary operators
auto val = [](auto n) { return [n]() { return n; } }; // returns a lambda
std::map<char, std::function<int()>> zeroOps { // nullary operators
    {'0', val(0)}, {'1', val(1)}, {'2', val(2)}, {'3', val(3)}, {'4', val(4)},
    {'5', val(5)}, {'6', val(6)}, {'7', val(7)}, {'8', val(8)}, {'9', val(9)},
};

std::map<char, std::function<void(std::vector<int>&)>> stackOps {
    {' ', [](auto &stack) { } }, // no operation
    {'c', [](auto &stack) { stack.clear(); } }, // Clear the entire stack
    {':', [](auto &stack) { // swap the top two elements
        auto top = stack.back(); stack.pop_back();
        auto second = stack.back(); stack.pop_back();
    } }
};
```

```
        stack.push_back(top);
        stack.push_back(second);
    } },
    { '=', [](auto &stack) {                // print the entire stack
        for(int elem : stack) { std::cout << elem; }
        std::cout << "\n";
    } },
};

void calculator(std::string input) {
    std::vector<int> stack {};
    for(char c : input) {
        int top, second;
        if(auto it = unOps.find(c); it != unOps.end()) {
            // if unary operator ...
            auto func = it->second;
            top = stack.back(); stack.pop_back(); // ... get top element
            stack.push_back(func(top));           // ... apply func, push result onto stack
        } else if(auto it = binOps.find(c); it != binOps.end()) {
            // if binary operator ...
            auto func = it->second;
            top = stack.back(); stack.pop_back(); // ... get the top 2 elements
            second = stack.back(); stack.pop_back();
            stack.push_back(func(second, top)); // ... apply func, push result onto stack
        } else if(auto it = zeroOps.find(c); it != zeroOps.end()) {
            // if nullary operator ...
            auto func = it->second;
            stack.push_back(func());             // ... result of func on stack
        } else if(auto it = stackOps.find(c); it != stackOps.end()) {
            // if stack operator
            auto func = it->second;
            func(stack);                         // ... apply func to stack
        } else {
            std::cout << "\n'" << c << "' I don't understand.\n";
        }
    } /* for c */
}

int main(int argc, const char* argv[]) {
    if(argc > 1) {
        calculator(argv[1]);
    } else {
        // 3+4*5+6 with multiplication before addition results in 29
        calculator("345*+6+=");
    }
}
```

```

calculator("93-=");           // 9 - 3 = Output: 6
calculator("82/=");           // 8 / 2 = Output: 4
calculator("92%=");           // 9 % 2 = Output: 1
}

```

Listing 28.43 This calculator maps keys to functors.

28.7.2 Function Generators

You have seen `function` used extensively in [Listing 28.43](#). It is simply the type of all callable objects in C++—almost. In fact, `function` is just something that can implicitly convert to a C function pointer and has various initialization options. It thus acts as a bridge between all callable objects and C function pointers. You have already read quite a bit about this in [Chapter 7](#).

Otherwise, there are a handful of helper functions in the vicinity of `function`, such as binding parameters.

Predefining Parameters with “bind”

If you have a function with two parameters and the first one is already fixed, you can use `bind` to make it a function with one parameter:

```

// https://godbolt.org/z/e6xT638ch
#include <functional> // subtract, minus, bind
#include <iostream>
using std::cout;
int subtract(int a, int b) { return a - b; }
int main() {
    using namespace std::placeholders;
    cout << subtract(9, 3) << '\n' ; // Output: 6
    auto minus3 = std::bind(subtract, _1, 3);
    cout << minus3(9) << '\n';      // Output: 6
    auto from9 = std::bind(subtract, 9, _1);
    cout << from9(3) << '\n';      // Output: 6
    auto againMinus3 = std::bind(std::minus<int>{}, _1, 3);
    cout << againMinus3(9) << '\n'; // Output: 6
}

```

Here, the strange `_1`, `_2`, and so on *placeholders* are important. They are defined in namespace `std::placeholders` and are intended to gather the arguments for the function. You insert such a placeholder where you do not yet know the value of the argument *now* because it will only be determined *later* at the call. If you create a function with one argument like `minus3`, you need to specify `_1`. A function with two parameters needs `_1` and `_2`.

If you create a function that does not require any parameters, then no placeholder is needed. A typical example is random generators, as in the `bind` variant in [Listing 28.44](#).

```
// https://godbolt.org/z/j7sa9rexh
#include <random>
#include <vector>
#include <iostream>
#include <functional>
void rollDice() {
    std::default_random_engine engine{};
    std::vector<size_t> counts{0,0,0,0,0,0};
    std::uniform_int_distribution<int> d6{0, 5}; // uniformly distributed integers
    auto d = std::bind(d6, engine);           // d() = d6(engine)
    for(auto i=1200*1000; i>0; --i) ++counts[d()];
    for(auto c : counts) std::cout<<" "<<c;
    std::cout << '\n';
}
int main() {
    rollDice();
}
```

Listing 28.44 With “`bind`”, you can also set all parameters of a function.

Lambda, “`bind`”, “`bind_front`”, and “`bind_back`”

Starting from C++20, there is the `bind_front` function and from C++23 also `bind_back`. These make writing some generic functions even easier.

The syntax of `bind` and `bind_back` is somewhat confusing for some. A lambda is often easier to read:

```
int add(int a, int b) { return a + b; }           // given this function
auto add_to_9_lambda = [](auto x) { return add(9, x); }; // as a lambda
auto add_to_9_bind = bind(add, 9, _1);           // with bind
cout << add_to_9_lambda(3) << '\n';              // Output: 12
cout << add_to_9_bind(3) << '\n';               // Output: 12

auto add_to_3_bind_back = bind_back(add, _1, 3); // with bind_back
auto add_to_3_lambda = [](auto x) { return add(x, 3); }; // as a lambda
cout << add_to_3_lambda(9) << '\n';             // Output: 12
cout << add_to_3_bind_back(9) << '\n';          // Output: 12
```

The `add_to_9_lambda` function does the same thing as `add_to_9_bind`. They bind a fixed value to the first parameter of the `add` function. What remains is a function with one parameter, which you can then call later in the program, as is done here with 3.

The `add_to_3_bind_back` function does the same thing as `add_to_3_lambda`. But they bind to the second parameter of the `add` function.

In generic programming, lambdas have difficulties with `noexcept` and `forward`. For these cases, you can resort to `bind`, `bind_front`, and `bind_back`.

Member Access with “`mem_fn`”

With `mem_fn`, you turn a method or a data member into a function. You then no longer use the following syntax:

```
instance.method(arg);
instance.data;
```

Instead, you write this:

```
method(instance, arg);
data(instance);
```

The following listing shows a more detailed example in which the `Numbers` class instance is made a parameter of the generated function object.

```
// https://godbolt.org/z/GMjcjrvaT
#include <functional>
#include <iostream>
struct Numbers {
    int theNumber() {
        return 42;
    }
    int more(int n) {
        return n + data;
    }
    int data = 7;
};
int main() {
    auto func = std::mem_fn(&Numbers::theNumber);
    auto func2 = std::mem_fn(&Numbers::more);
    auto access = std::mem_fn(&Numbers::data);
    Numbers numbers;
    std::cout << func(numbers) << '\n';           // Output: 42
    std::cout << func2(numbers, 66) << '\n';       // Output: 73
    std::cout << access(numbers) << '\n';         // Output: 7
}
```

Listing 28.45 This is how you turn class members into free functions.

Most of the time, you can use `mem_fn` instead of `bind`, but the magic of turning a data member into a function cannot be done directly by `bind`. And the same applies here: lambdas are possibly more expressive and easier to read.

28.8 “optional” for a Single Value or No Value

In C++, `optional<T>` has a slightly different meaning than in Java. There, the “generic” `Optional<T>` is part of the streaming API and thus represents a “container of size one or zero.” It is recommended not to use it as a parameter for optional values there.

This is different in C++: First, it is *not* a container that works with algorithms. Second, `optional<T>` is specifically designed to indicate, as a parameter and return value or in containers, that a value may or may not be present.

The documentation states the intended use: “It is recommended to use `optional<T>` in situations where there is exactly one clear reason not to have a value of type `T`, and where the absence of the value seems as natural as assuming a regular value.” Less formally and shorter, but also less precise: Where you are tempted to use a `nullptr` to indicate the absence of a value, you should use `optional<T>`.

The uses of an `optional<T> opt` are as follows:

- You create it via constructor, either empty or with a regular value, the special value `nullopt`, or you use `make_optional`.
- You use the implicit conversion to a `bool` to check if a value is present, so `if(opt)`, or you use `has_value()`.
- You access the value via `value()`, `value_or(fallback)`, `*opt`, or `opt->`.
- Replace the old value via assignment, `swap`, or delete it with `reset()`.
- Compare with `==`, `<=`, and so on. It is possible to compare with the empty value.
- Chain from C++23 with `transform`, `and_then`, and `or_else`.

`optional` is not well-suited for storing `bool`s and pointers. For a three-valued Boolean type, it is better to use `boost::tribool`, and pointers already have a nonvalue with `nullptr`.

28.9 “variant” for One of Several Types

With this template, you can model something similar to a `union`, but in a safer form.

A `union` has the problem in safe programming that constructors or destructors of the contained objects are not reliably called.

A variant can take the value of one of several types. Setting a value of one type changes the state of variant, so attempting to read another type from it then fails. Thus, its usage is safe.

```
// https://godbolt.org/z/baY7Tz91f
#include <variant>
using std::get;
int main() {
    std::variant<int, float> v{};
    v = 12; // State changes to int
    auto i = get<int>(v); // retrieves the int
    std::cout << i << '\n'; // Output: 12
    v = 3.456f; // State changes to float
    std::cout << get<float>(v) << '\n'; // Output: 3.456
    get<double>(v); // ✗ Error
    get<3>(v); // ✗ Error
    std::variant<int, float> w{};
    w = get<float>(v); // Access by type
    w = get<1>(v); // Access is also possible via index
    w = v; // entire assignment is also possible
    try {
        get<int>(w); // triggers exception
    } catch (std::bad_variant_access&) { /* ... */ }
}
```

Listing 28.46 Basic functionalities of “variant”.

Here I declare `v` as a variant that can contain either a value of type `int` or one of type `float`. Then I assign `v` with 12 an `int`, which I then read and output with `get<int>`. This is followed by another assignment with a value, but this time a `float`. I also read and output this one. `v` has thus contained first an `int` and then a `float` in succession.

You cannot request a type with `get` that variant does not know. `get<double>` does not compile.

At runtime, you will get an error if you query a valid type that variant does not currently contain. You will receive the `bad_variant_access` exception.

Without an exception, you check the state with `holds_alternative<int>(w)`. Here you get a `bool` indicating whether `w` currently holds an `int` or not.

If you want to react to the content of variant, you can query which one is current with `index`. However, it is more elegant to use `visit`, to which you pass a *visitor* that can handle the different types.

```
// https://godbolt.org/z/G4z8Y5E6P
#include <variant>
#include <iostream>
using std::cout;
struct TypeGreeting {
    void operator()(int) const { cout << "Hello int"; }
    void operator()(float) const { cout << "Hello float"; }
};
int main() {
    std::variant<int, float> var{};
    var = 12; // State int
    std::visit([](auto a) { cout << a; }, var); // generic lambda
    cout << std::endl;
    var = 3.456f; // State float
    std::visit(TypeGreeting{}, var); // Functor with overloads
    cout << std::endl;
}
```

Listing 28.47 Inspecting a “variant” using a visitor.

In the first visit, the lambda is generic and accepts any parameter that can be output to a stream.

The second use of `visit` takes a functor as a parameter, which has specific overloads for both possible types. The `operator()` methods for `int` and `float` could contain entirely different implementations.

28.10 “any” Holds Any Type

If you are dissatisfied with having to specify the possible types when declaring `variant`, then perhaps `any` is for you. You can store a value of any type in it. And you can only retrieve that type. If you try with a different type, you will get an exception. However, you can assign any a *new value* and also a *new type*:

```
// https://godbolt.org/z/1eco9Wcr3
#include <any>
#include <iostream>
#include <vector>
#include <string>
int main() {
    std::any a = 5;
    std::cout << std::any_cast<int>(a) << '\n';
    a = 3.456;
    std::cout << std::any_cast<double>(a) << '\n';
}
```

```

using namespace std::literals;
std::vector<std::any> data { 4, 8.976, "Geronimo"s };
std::cout << std::any_cast<double>( data[1] ) << '\n';
std::cout << data[1].type().name() << '\n';
}

```

You see how I first assign an `int` and then a `double`. Instead of using `get`, you retrieve the value here with `any_cast`, which is a slightly different mechanism. However, if you ask for the wrong type, you will get an exception.

You can also pack any into containers. The three any elements in `data` have different types after initialization. For information, you can also access the currently stored type with `type()`. However, this is more for debugging and is not portable information.

Note the following points:

- `any` is not a template class.
- You create any via constructor with or without value or `make_any`.
- You check if an any is empty with `has_value()`.
- You change the value with an assignment, `emplace`, `reset`, or `swap`.
- You retrieve the value type-safely with `any_cast<T>`.
- There is no exception-free check to see if an any currently contains a specific type. If you need that, consider `variant`.

28.11 Special Mathematical Functions

The mathematicians and physicists among you will be pleased that the standard library already includes some (really) very special mathematical functions, so you don't need to use a third-party library. Most of the functions operate on `float`, `double`, and `long double`. Sometimes there are multiple name variants; I only mention one. The functions are located in the `<cmath>` header.

- **laguerre, assoc_laguerre, legendre, assoc_legendre, hermite**
Calculates (associated) Laguerre polynomials, (associated) Legendre polynomials (spherical functions), and Hermite polynomials.
- **beta, expint**
Calculates the Euler beta function and the exponential integral function.
- **comp_ellint_1, comp_ellint_2, comp_ellint_3, ellint_1, ellint_2, ellint_3**
Calculates the complete or incomplete elliptic integral of the first, second, and third kind.
- **cyl_bessel_i, cyl_bessel_j, cyl_bessel_k, sph_bessel, sph_legendre, sph_neumann**
Regular, nonregular, and irregularly modified cylindrical Bessel functions as well as the spherical Bessel function, Legendre function, and Neumann function.

- **cyl_neumann**
Cylindrical Neumann function, also called the Bessel function of the second kind.
- **riemann_zeta**
Calculates the Riemann- ζ function, which you might need in connection with prime numbers.

C++20: Mathematical Constants

Starting with C++20, you have access to the most interesting mathematical constants. In the `<numbers>` header, the following inline constexpr double are defined in the `std::numbers namespace`: `e`, `log2e`, `log10e`, `pi`, `inv_pi`, `inv_sqrtpi`, `ln2`, `ln10`, `sqrt2`, `sqrt3`, `inv_sqrt3`, `egamma`, and `phi`.

If you want the float or long double value of one of these constants, append the `_v` suffix to the name for a template—for example:

```
pi_v<long double>
```

28.12 Fast Conversion with “<charconv>”

There are many ways to convert a string (or `char*`) into a number. With C++17, there is another option: `from_chars` from the `<charconv>` header. The main focus of `from_chars` is on performance, as it becomes increasingly important with the growing number of XML and JSON applications.

`from_chars` comes with the conversion function `to_chars` for converting numbers to strings.

`from_chars` shows its strengths when you have a `char*`, already know that you need to parse a number, and have nothing to do with locales (different decimal and thousand separators) and encodings (UTF-16, etc.).

The existing ways to convert a string to an int or the like include the following:

- **sscanf**
From C; flexible, related to `printf`.
- **atol, atoi, atoll, atof**
From C; no way to determine the end or success of parsing.
- **strtol, strtod, strtod, strtold, strtoll, strtoull**
Good candidate; error checking is okay, but lacks overflow and underflow checking.
- **stringstream**
C++; flexible, but slow, also lacks overflow and underflow checking.
- **stol, stoi, stoll, stoull, stof, stod, stold**
C++11; lacks flexible start position, error handling via exception.

And since C++17 in the assortment:

```
from_chars_result
```

```
    from_chars(const char* first, const char* last, T& value, int base = 10);
```

The type `T` can be any integer or floating-point type. The result is read into the value output parameter. Thus, the variable must be declared beforehand.

The return is a struct with the `const char*` ptr and `errc` elements. ptr indicates how much was read and `errc` shows any potential error. It's nice that C++17 also allows structured binding:

```
auto [pbis, errc] = std::from_chars(pbegin, pend, value);
```

However, `from_chars` does not handle leading whitespaces. These must be skipped beforehand:

```
// https://godbolt.org/z/115nE7xsc
#include <charconv>
#include <vector>
#include <iostream>
#include <string>

std::vector<size_t> num_to_vec(const std::string& nums) {
    std::vector<size_t> result {};
    // without trailing spaces
    const auto end = nums.data() + nums.find_last_not_of( ' ' ) + 1;
    const char* st = nullptr; // loop pointer
    auto last = nums.data(); // last untranslated character
    size_t n; // converted number
    do {
        for(st = last; (st<end)&&(*st==' '); ++st); // skip ''
        if (last = std::from_chars(st, end, n).ptr; last != st)
            result.push_back(n); // store number
    } while (last != st);
    return result;
}

void errorDemo(const char* buf, size_t sz) {
    int n;
    auto [p, ec] = std::from_chars(buf, buf+sz, n);
    if (ec != std::errc{}) {
        const auto error = std::make_error_code(ec);
        std::cout << error.message() << '\n';
    }
}
```



```
int main() {
    auto result = num_to_vec("12 33 43");
    for(auto r : result) std::cout << r << " ";
    std::cout << '\n';
    // Output: 12 33 43
    errorDemo("XYZ", 4);
    // Output: Invalid argument
    errorDemo("123123123123123", 16);
    // Output: Numerical result out of range
}
```

First, we find the end of the string and ignore any whitespace there. In the loop, we also ignore all whitespace, as `from_chars` would not do that. The result ends up in `n`. In `errorDemo`, a conversion error is also handled. In `p`, we have the position where the conversion stopped.

Experiments show that `from_chars` can be very fast. Depending on requirements and tests, this naturally varies, but one source states: if `from_chars` takes one unit of time, `strtoul` and `atoi` take about two, `stoul` about four, and `stringstream` about six units of time. `from_chars` was particularly fast in cases only the first number needed to be extracted from a string.⁹ Other experiments are even more in favor of `from_chars`.¹⁰

The `to_chars` function works very similarly. It also returns a pointer and error code in a struct and writes the result to an output parameter:

```
to_chars_result to_chars(char* first, char* last, T value, int base = 10);
```

Here, structured binding helps again:

```
// https://godbolt.org/z/dcqenEd3l
#include <iostream>
#include <charconv>
#include <string_view>
#include <array>
int main() {
    std::array<char, 10> str {};
    if(auto [p, ec] = std::to_chars(str.data(), str.data() + str.size(), 42);
        ec == std::errc{} )
        std::cout << std::string_view(str.data(), p - str.data()) << "\n";
}
```

The result ends up in `str`. For this, `to_chars` needs the beginning and the end of the range into which it can write.

9 *Floating-Point <charconv>: Making Your Code 10x Faster With C++17's Final Boss*, https://www.youtube.com/watch?v=4P_kbFOEbZM, Stephan T. Lavavej, 2019

10 *How to Use The Newest C++ String Conversion Routines - std::from_chars*, <https://www.cppstories.com/2018/12/fromchars/>, Bartłomiej Filipek, 2021-06-16, [2024-08-15]

Chapter 29

Threads: Programming with Concurrency

Chapter Telegram

- **Thread**

A “strand” of program execution that can run concurrently with other threads.

- **Semaphores**

Mechanism to limit concurrent access to a shared resource. A semaphore exists per shared resource or group of resources.

- **Mutex**

Semaphores that allow access to exactly one thread.

- **Lock**

Group of classes that define the range in which access should be protected using a mutex. A lock exists per thread and resource.

- **Critical section**

Program piece protected from concurrent execution using lock and mutex.

- **Data race**

Condition where multiple threads unpredictably modify data.

- **Deadlock**

Condition when two threads holding locks on mutexes block each other, and neither can proceed.

- **Promise and future**

Model of a communication channel between threads. A promise is the commitment to deliver a result via a future when requested.

- **Memory order**

This specifies the temporal relationship in which read and write operations in different threads on an object must occur.

This chapter introduces you to the parts of the standard library that deal with *multi-threading*. In doing so, I will also explain a few principles around this type of concurrent programming so that you know how to use the classes and functions correctly.

So far, we have considered a program as a sequence of instruction to be executed sequentially. With branches and functions, the computer can jump back and forth, but it has always executed one instruction after another. For some time now, computers

have had multiple processing units and thus the ability to work on different parts of a program simultaneously. Your program can work *in parallel*. These parallel execution threads are called *threads*.

All running threads share the program's memory. All threads can see all the memory of all other threads—that is, all variables and values. And unfortunately, not only do they see, but they can also write to them if they want. The trick is to do this in an orderly manner so that the data remains consistent.

Other Forms of Parallelism

There are other forms of parallelism—all that you can imagine. For example, there can be a thread that works on multiple data simultaneously—effectively working on the same memory with the same instructions (SIMD from graphics cards). It is also parallel when a single CPU instruction multiplies eight pairs of numbers simultaneously (MMX, SSE, and AVX). You can also achieve parallelism by running a program distributed across one or more computers and having them communicate with each other (processes, sockets, MPI, and others).

All these forms are not yet covered in the C++ standard and therefore are not part of this book. I limit myself to the execution of *a* program with *one* memory with *multiple* threads.

With C++17, the possibility for SIMD for many algorithms has been added. It is listed under `parallel_unsequenced_policy`. You then specify the additional `par_unseq` parameter. If you are interested, check what your compiler already supports and what it can do with it.

29.1 C++ Threading Basics

You have already started threads, even if you didn't realize it, because in every C++ program, at least the main thread exists. For you, this means that upon entering `main`, you are in your first thread, which the system has started for you. When you leave `main`, the main thread terminates after some mandatory cleanup.

You achieve multithreading when you start additional threads. Each thread has an entry and exit point, just like `main` for the main thread. Anything callable can define this entry and exit, such as a function, a functor, a lambda, a bound method, a `function<>` instance—simply anything that can implicitly convert to a function pointer.

In C++, a thread is an instance of type `std::thread` from the `<thread>` header. With this instance, you can interact a bit, such as waiting for its termination, called *joining*. If you don't join, but the thread is still running, `terminate()` is called, thus ending your program. Alternatively, you can also “detach” the thread by calling `detach()`. Then your thread would continue running, but the program is terminated. This is usually not

what you want. Therefore, you should always wait for the termination before ending the program.

Hence, since C++20, there is the *joining thread* with the `jthread` class. These threads attempt to terminate themselves in the destructor and then wait. Normally, you will use `jthreads`. Everything said in this chapter essentially applies to both `thread` and `jthread`. If you need to use `thread` instead of `jthread`—for example, because C++20 is not available and `jthread` is not part of your version of the standard library—then you must not forget to `join()` the thread. I point out further differences between `jthread` and `thread`. If you still don't want to do without `jthread`, you can use the library at <https://github.com/josuttis/jthread> by Nico Josuttis, the original author of the `jthread` class. It is compatible with C++17.

But before you can wait for the termination of a thread, you must of course first start a new thread or `jthread`.

29.1.1 Starting Pure Threads

For this, you need a *function object* (I will use this term from now on to represent anything that can be started by a thread) and an existing thread that starts the new one. The latter is easy—you have `main`.

```
// https://godbolt.org/z/qnadYPWh1
#include <iostream>
#include <thread>
using std::cout; using std::endl;

long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
void task1() { auto r = fib(40); cout << "fib(40)=" << r << endl; }
void task2() { auto r = fib(41); cout << "fib(41)=" << r << endl; }
void task3() { auto r = fib(42); cout << "fib(42)=" << r << endl; }

struct BackgroundTask {
    void operator()() const {
        task1();
        task2();
        task3();
    }
};

int main() {
    BackgroundTask backgroundTask{}; // Initialization, does not compute anything yet
    std::jthread myThread{ backgroundTask }; // Computation starts
}
```

Listing 29.1 How to start a thread.

You start a thread by creating a `std::jthread` object. As a constructor argument, you then pass something callable, here an instance of the `BackgroundTask` functor.

Note that the callable object is always *copied* for the new thread. Therefore, the callable object must be sensibly copyable. If you use `backgroundTask` for multiple `jthread` instances, they do not share the same instance.¹

It is quite common to create the callable object directly in the constructor call of `jthread`:

```
std::jthread myThread{ BackgroundTask{} }; // temp-value as constructor argument
```

Be careful not to write `BackgroundTask()` here, as that does not create a temp-value but declares a function. You can completely avoid this problem by writing simple tasks as lambda expressions when you initialize the `jthread`:

```
// https://godbolt.org/z/nPGY47r4K
std::jthread myThread{ [] {
    task1();
    task2();
    task3();
} };
```

This means that `myThread` starts a new thread that runs in parallel. In that thread, the `task1`, `task2`, and `task3` functions are then executed sequentially.

29.1.2 Terminating a Thread Prematurely

It is often the case that you do not want a thread to run until it finishes on its own. For example, perhaps the **Cancel** button was clicked while the thread is reading a huge file from the disk. It would be undesirable if the thread then continued to read the file to the end, even though the data is no longer needed.

The solution for this is `stop_token`, which is already built into `jthread`. In thread, this does not exist yet, so you have to implement it yourself.

The `stop_token` is an object that you get from `jthread` and then use as a communication channel into the thread. In the thread, you should occasionally check whether the outer thread requests premature termination.

```
// https://godbolt.org/z/6cqd7Tsae
struct BackgroundTask {
    void operator()(std::stop_token st) const { // Token for communication
        task1();
        if(st.stop_requested()) return;
        task2();
    }
};
```

¹ Unless you take further measures.

```

        if(st.stop_requested()) return;
        task3();
    }
};

int main() {
    BackgroundTask backgroundTask{};
    std::jthread myThread{ backgroundTask };
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // wait 100ms
    myThread.request_stop(); // request the thread to stop
}

```

Listing 29.2 With “stop_token”, the outside world communicates into the thread.

Instead of outputting the result of all three tasks, only the result of the first task is output here.

When a `jthread` is created, the class always also creates a `stop_source`. From this `stop_source`, a `stop_token` is passed into the called function as a parameter if the method has a parameter of this type. If not, then not. So it is, so to speak, an optional parameter of your actual thread function.

In this case, the thread function received it as `st`. In between, we periodically check with `st.stop_requested()` to see if someone wants the thread to terminate itself. If that is the case, we stop the work and clean up properly. And that is important! In C++, you cannot simply terminate a thread from the outside using standard means. That would break the RAII concept, because, for example, destructors would no longer be called.

From the outside, you get the `stop_source` object with `get_stop_source()` and then call `request_stop()`. Or you use the shortcut to directly call `request_stop()` on the `jthread`, as in this example.

It is really convenient with `jthread` that it can flexibly handle thread functions that have a `stop_token` as a parameter, but also those that do not. However, you should note that without such a parameter, you cannot check if the thread should be terminated.

For simplicity, I have mostly omitted `stop_token` in this chapter. But you should always include it when writing a thread that runs for more than a few milliseconds.

29.1.3 Waiting for a Thread

Normally, you don't want to terminate a thread prematurely but wait until it finishes. Once you have started a thread, you need to do one of the following:

- **Wait until the thread finishes**

You do this by joining the thread. This happens automatically with `jthread` in the destructor.

- **Let the thread run independently**

You need to detach from the newly started thread.

If you haven't decided before the starting thread ends and use `thread`, your entire program will terminate. Specifically, this means that if the new thread is still running but the destructor of `thread` is called, it will invoke `std::terminate()`—and thus abruptly end your program:

- **Wait**

```
myThread.join()
```

- **Detach**

```
myThread.detach()
```

So let's use `thread` instead of `jthread` and make the waiting explicit:

```
// https://godbolt.org/z/Tzs93Mzfk
std::thread myThread{ [] { // pure thread
    task1();
    task2();
    task3();
} };
myThread.join(); // waits for the thread to finish
```

Once again: If the scope of the `thread` variable is exited and the thread is still running and you have not detached it with `detach()`, your program will basically crash. The call to `join()` waits for the thread to finish.

In this simple case, `join()` is the only correct solution. Because if the main program terminates otherwise, detached threads will also be terminated. So if you use `detach()` here, you will not see any output because `main` will exit shortly after, before the calculations were finished. In another function, however, you can certainly use `detach()`, as there are likely many more calculations taking place.

29.1.4 Consider Exceptions in the Starting Thread

Note that you also need to consider exceptions: `myThread` will also be removed if an exception is thrown somewhere in the calling thread causing your `thread` variable to go out of scope. If the started thread is not yet finished, your entire program will be terminated. Thus, an exception that you handle one level too far outside may have significant consequences:

```
// https://godbolt.org/z/E49on6zn3
#include <iostream>
#include <thread>
#include <vector>
#include <exception>
using std::cout; using std::endl;

long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
```

```

void task1() { auto r = fib(40); cout << "fib(40)=" << r << endl; }

void main_program() {
    try {
        std::thread th{ &task1 };
        std::vector data{ 0,1,2 };
        data.at(666);                // ✗ triggers out_of_range
        th.join();                    // would wait
    } catch(std::runtime_error &ex) { // ✗ does not match out_of_range
        /* ... */
    }
}

int main() {
    try {
        main_program();
    } catch( ... ) {                  // so far, so good, looks safe
        std::cout << "An error has occurred\n"; // you won't see this
    }
}

```

In the absence of threads, this would be working code. `data.at(666)` does throw an exception, but that's not forbidden. The thrown `out_of_range` exception does not match the `runtime_error` specified in `catch`. So the exception leaves `main_program` and is caught in `catch(...)`. There—without threads—the output “An error ...” would appear.

Not so with a still running thread: `th` is still active at the time of the exception. For exception handling, its block is exited and the destructor of `thread` is called. And that then leads to `terminate()`, the immediate program termination.

What if you had already caught everything possible with `catch(...)` instead of `catch(std::runtime_error ex)` in the main program? Unfortunately, there's no difference: the scope of `th` is still exited when handling the exception in `catch`. You need to ensure that `th` still exists within `catch`—for example, like this:

```

// https://godbolt.org/z/T4c1GnjY4
#include <iostream>
#include <thread>
#include <vector>
#include <exception>
using std::cout; using std::endl;

long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
void task1() { auto r = fib(40); cout << "fib(40)=" << r << endl; }

```



```
void main_program() {
    std::thread th{ &task1 };
    try {
        std::vector data{ 0,1,2 };
        data.at(666);           // ✖ triggers out_of_range
    } catch(std::runtime_error &ex) { // does not match out_of_range
        /* ... */                // handle specific error here
    } catch( ... ) {
        th.join();
        throw;                    // continue error handling outside
    }
    th.join();                    // waits after Okay or specific error
}

int main() {
    try {
        main_program();
    } catch( ... ) {
        std::cout << "An error has occurred\n";
    }
}
```

The lifetime of the `th` instance is now separate from the computations that can go wrong. In the example, I distinguish between the two types of exceptions: “I need to do something locally specific here” with `std::runtime_error`, and everything else ..., which I do not expect locally, but know or assume will be handled further outside—which is why I rethrow with `throw`; after the important `th.join()` to the outer handling. I call `join()` as late as possible because the call waits until the computation in the thread is finished (the call blocks). If I were to do it earlier, such as right after the start, I wouldn't get parallel execution.

This is much simpler with `jthread`! Because the waiting happens in the destructor itself, this is also the case with an exception:

```
// https://godbolt.org/z/8TdzEGf3d
void mainProgram() {
    std::jthread th{ &task1 };
    std::vector data{ 0,1,2 };
    data.at(666);           // ✖ triggers out_of_range
}
```

The `main()` function remains the same. The `jthread` is executed until the end despite the exception from `data.at(666)` because the destructor of `th` contains a `join()`. So you first see `fib(40)=102334155` and then `An error has occurred` on the console.

The Destructor of “jthread”

The destructor of `jthread` does two things:

- It calls `request_stop()` on the `stop_token` of the thread function.
- It waits for the end of the thread function with `join()`.

This means that when your `jthread` variable, such as `th`, goes out of scope, its destructor is called. If your thread function does not have a `stop_token` as a parameter, the thread will continue to run until the end, even though it probably should be stopped.

Therefore, you should always give your thread function a `stop_token` as a parameter and occasionally check with `stop_requested()`. Even if you never call `request_stop()` from outside, the destructor does.

29.1.5 Passing Parameters to a Thread Function

It would be very impractical if you had to write a separate function for each different call to `fib` that sets the parameter `n` and then calls `fib`. You can also pass the parameters directly when starting the thread. To do this, simply add the desired parameters to the initializer of thread.

```
// https://godbolt.org/z/bMdrWrd4c
#include <iostream>
#include <thread>
using std::cout; using std::endl;

long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
void runFib(long n) {
    auto r = fib(n);
    cout << "fib("<<n<<"")=" << r << endl;
}

long ack(long m, long n) { // Ackermann function
    if(m==0) return n+1;
    if(n==0) return ack(m-1, 1);
    return ack(m - 1, ack(m, n-1));
}

void runAck(long m, long n) {
    auto r = ack(m, n);
    cout << "ack("<<m<<','<<n<<"")=" << r << endl;
}

int main() {
    std::jthread f40{ runFib, 40 };
    std::jthread f41{ runFib, 41 };
}
```

```

std::jthread f42{ runFib, 42 };

f40.join(); f41.join(); f42.join();

std::thread a1{ runAck, 4, 0 };
std::thread a2{ runAck, 4, 1 };
std::thread a3{ runAck, 2, 700 };
std::thread a4{ runAck, 3, 10 };
}

```

Listing 29.3 Add parameters to the thread function in the constructor.

You can simply pass one or more parameters additionally to the constructor of `jthread`.

Ackermann and Fibonacci

The `fib(long n)` function for the n -th Fibonacci number I have used often in different variants in this book as an example for a calculation. The n -th Fibonacci number is the sum of the $n - 1$ -th and $n - 2$ -th Fibonacci numbers. The first ones are 1, 1, 2, 3, 5, 8, 13, 21—and the sequence continues.

I like to use it because first, you can write the function in one line, and second, it keeps the computer busy for a while. For `fib(40)`, the computer already needs a few seconds to compute; `fib(45)` becomes a test of patience. However, this is only the case if you write it—as I do here—naively *recursively*. With a little more effort, you can also get an efficient function, as shown in [Listing 29.3](#). But I don't want to do that for the thread examples.

Even more extreme is the *Ackermann function*. It has two parameters and grows extremely fast, especially when the first parameter `m` is increased. It was conceived to demonstrate the limits of automatic calculations, and for this reason, it is well-suited for my threading examples. Its values are less illustrative than Fibonacci numbers and, apart from its rapidly growing size, less interesting.

Ahead, I will assume the definitions of the functions `ack` and `runAck` to save space.

Parameters Are Always Copied

The parameters are copied into the thread before it is started. So, a reference is not stored. Thus, new strings are created for each thread.

```

// https://godbolt.org/z/fbnnKzrs1
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono; // seconds, suffix s

```

```

void delayPrint(seconds s, const std::string& msg) {
    std::this_thread::sleep_for(s);
    std::cout << msg << std::endl;
}

int main() {
    std::jthread m1{ delayPrint, 1s, "On your marks" };
    std::jthread m2{ delayPrint, 2s, std::string{"set"} };
    std::string go = "go";
    std::jthread m3{ delayPrint, 3s, go };
}

```

Listing 29.4 Parameters are copied into the thread.

Although `go` could be referenced as a local variable *and* the parameter `msg` of `delayPrint` is a `const string&`, `go` is still copied first before being passed to the thread function `delayPrint`. This ensures that even if you call `detach()` on `m3` and leave its scope, the potentially important reference parameter of the thread function still exists.

As usual, you need to be more careful here when working with raw pointers. Because here only the pointer is copied before it is passed to the thread function, but of course not what it points to. This is going to end badly.

```

// https://godbolt.org/z/hvq5dbzde
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono; // seconds, suffix s

void delayPrint(seconds s, const char* msg) { // ⚡ raw pointer
    std::this_thread::sleep_for(s);
    std::cout << msg << std::endl;           // ⚡ this won't work
}

void run() {
    const char risk[] = "This won't end well...";
    std::jthread t{ delayPrint, 1s, risk }; // ⚡ raw pointer
    t.detach();
    // here the scope of 'risk' is left
}

int main() {
    run();
    std::this_thread::sleep_for(2s);         // wait another 2 seconds
}

```

Listing 29.5 Use caution with raw pointers as parameters.

Here, `risk` is destroyed as soon as `run()` is exited. The `msg` pointer still points to its old location and thus to an invalid area; undefined behavior is the result.

Enforcing Reference Parameters

In some cases, however, you want to have a reference, and it would be impractical or impossible to copy the parameter. Then use `std::ref` to enforce a reference.

```
// https://godbolt.org/z/GnsWYvecv
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono; // seconds, suffix s
struct State {
    int counter;
};
void showState(const State& state) {
    for(auto i : { 5,4,3,2,1 }) {
        std::cout << "counter: " << state.counter << std::endl;
        std::this_thread::sleep_for(1s);
    }
}
int main() {
    State state { 4 };
    std::jthread th{showState, std::ref(state)}; // remains reference to state
    std::this_thread::sleep_for(1s);
    state.counter = 501;
    std::this_thread::sleep_for(1s);
    state.counter = 87;
    std::this_thread::sleep_for(1s);
    state.counter = 2;
}
```

Listing 29.6 Forcing a reference with `ref`.

The local `state` variable is passed here as a reference to the new thread and remains so. No copy is created, as you can see from the fact that changes to `state.counter` in `main` also affect `showState`. Otherwise, the output would consistently be `counter:4` and not as shown here:

```
counter: 4
counter: 501
counter: 87
counter: 2
counter: 2
```

Parameter by Move

Instead of a copy or reference, assume you have a resource that you cannot or do not want to copy, but you can move—for example, a `unique_ptr` (copying forbidden) or an `Image` you painted (copying expensive). In that case, you use `std::move`, and the source object is empty after starting the thread.

```
// https://godbolt.org/z/Pjzqsv9nz
// ... includes ...
#include <thread>
using namespace std::chrono; // seconds, suffix s
struct Image {
    std::vector<char> data_; // Copying expensive
    explicit Image() : data_(1'000'000) {}
};
void showImage(Image img) {
    std::cout << img.data_.size() << '\n';
}
void showIptr(std::unique_ptr<int> iptr) {
    std::cout << *iptr << '\n';
}
int main() {
    // expensive to copy, but good to move:
    Image image{};
    std::cout << image.data_.size() << std::endl;    // Output: 1000000
    std::jthread th1{ showImage, std::move(image) }; // Output: 1000000
    std::this_thread::sleep_for(1s);
    std::cout << image.data_.size() << std::endl;    // Output: 0
    th1.join(); // explicitly wait until the thread is done
    // impossible to copy, but good to move:
    auto iptr = std::make_unique<int>( 657 );
    std::cout << (bool)iptr << std::endl;            // Output: 1 for true
    std::jthread th2{ showIptr, std::move(iptr) };    // Output: 657
    std::this_thread::sleep_for(1s);
    std::cout << (bool)iptr.get() << std::endl;      // Output: 0 for false
}
```

Listing 29.7 Using “move” to transfer inputs to the thread.

The `Image` class contains data representing a large actual image. I choose `vector` as the container for the data because it can be moved. And because I adhere to the *rule of zero*, the `Image` class is also movable.

Before I start `th1`, `image.data_` contains a lot of data. Due to `std::move` in the constructor of `th1`, the `image` or its content is *moved* into the thread. The local variable `image` in `main` is empty afterward, as indicated by the size 0.

It is similar with `iptr`, except that you cannot even copy a `unique_ptr`, even if you wanted to. Before creating `th2`, the pointer held by `iptr` is valid: `(bool)iptr` outputs 1, which stands for *true*. After moving at the start of `th2`, the same expression outputs 0 for *false*, which means that `iptr` no longer contains an `int`. It was moved into the thread function `showIptr`.

29.1.6 Moving a Thread

You do not need to keep `jthread` as a local variable. You can have a function or method create `jthread` instances—that is, return them as values.

```
// https://godbolt.org/z/Kv84eb96E
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono; // seconds, suffix s
auto makeThread(std::string who) {
    return std::jthread{ [who] {
        std::this_thread::sleep_for(1s);
        std::cout << "Good luck, " << who << std::endl;
    } };
}
int main() {
    auto th = makeThread("Jim"); // Output: Good luck, Jim
    th.join();
}
```

Listing 29.8 Returning a thread.

You cannot copy `jthread`. If you want to pass a thread as a parameter, you have to move it.

```
// https://godbolt.org/z/8cMTrEWz6
#include <thread>
#include <chrono>
using namespace std::chrono; // seconds, suffix s
void yourMission(std::jthread job) {
    job.join();
}
int main() {
    std::jthread th{ [] {
        std::this_thread::sleep_for(1s);
        std::cout << "should you choose to accept it" << std::endl;
    } };
}
```

```

    } };
    yourMission( std::move(th) ); // Transfer responsibility
}

```

Listing 29.9 Moving a thread.

The `th` instance in `main` is moved to the thread function. It would be fine to exit `main` and let `th` be removed. The responsibility has been transferred to the thread function. In this specific example, it is just waiting for the thread to finish anyway, but as far as the lifetime of `th` in `main` is concerned, `detach` would also be okay.

The most practical aspect of `jthread` is that containers can hold them. This way, you can manage a completely dynamic number of threads with little effort.

```

// https://godbolt.org/z/YrEPnYMaP
#include <iostream>
#include <thread>
#include <vector>
using std::cout; using std::endl;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
void runFib(long n) { auto r = fib(n); cout << "fib("<<n<<"")=" << r << endl; }
int main() {
    std::vector<std::jthread> threads;
    // starten
    for( auto n : { 38, 39, 40, 41, 42, 43, } ) {
        threads.emplace_back( runFib, n );
    }
}

```

Listing 29.10 Threads in containers.

Not even waiting with `join()` on the elements of the container is necessary because the destructor of `jthread` takes care of that.

As you know, `emplace_back` creates the threads directly in place in the vector, but you can also use one of these two variants:

```

std::jthread th1{runFib, n};
threads.push_back( std::move(th) );           // Moving a thread variable
threads.push_back( std::thread{runFib, n} ); // temp-value automatically moved

```

With `thread` instead of `jthread`, don't forget to join all threads in the container at the end using `join()`.

29.1.7 How Many Threads to Start?

How many threads can a program start and manage? How many threads can perform calculations simultaneously? The standard does not specify upper limits for either, but a physical upper limit exists.

There is a difference whether you start a thread that mainly does nothing and waits for some event, or if you run a thread for number crunching. The latter burdens and blocks CPU and memory, consuming limited resources. Even doing nothing is not clearly defined, as it can include activities like reading data from the hard drive or the internet: the CPU is so much faster than an SSD, hard drive, or Wi-Fi that the majority of time spent reading is actually waiting. I would like to informally distinguish these two categories here with the terms *computation threads* and *waiting threads*.

Waiting threads can be managed by the system in large numbers simultaneously—but only if they do not interfere with each other over a shared resource. Each thread requires some memory for itself, perhaps a megabyte, so you could practically have between 1,000 and 10,000 threads. In such situations, thread pools often provide a better alternative. But you can usually have 10 to 100 such threads on current machines without any problems. And here, by *normal machines*, I mean your desktop PC, laptop, or smartphone. Systems for special applications like cameras, cars, or networked LEDs (so-called embedded systems) might be somewhat more restricted.

Compute threads burden the CPU resource. Allowing more than one thread to compute on a CPU would be wasteful and unnecessary. But first, compute threads also need to wait from time to time and can yield their capacities to another thread during that time, and second, it is not really bad to have slightly more threads than CPUs. However, having significantly more threads than CPUs is not good because it leads to more thread switches than necessary. And each thread switch costs additional resources on top of what a thread itself costs. The rule of thumb is that you can run one to two times as many compute threads in parallel as you have CPU cores without overloading the system.

So if you want to know the optimal number of threads for a computational program, you first need to find out how many CPUs your machine has. This number times one or two is then the maximum number of computational threads I recommend. For this purpose, there is the following:

```
std::thread::hardware_concurrency()
```

This function provides a *hint* about the number of threads that can actually run in parallel on the system. However, the function can also return 0 if the information is not available.

Typically, you get the number of CPUs in the system. But even that is not guaranteed. Because nowadays, the matter of CPUs and threads is no longer so simple—not to mention special hardware (like graphics cards). Some CPUs already offer a kind of thread at

the hardware level: the more expensive Intel CPUs support *hyperthreading*. That means, for example, they have four *cores* (quasi-CPU), each of which can run two hyperthreads. Hyperthreading means that one thread can theoretically utilize a core at 100%, two threads as well, but from three onward the CPU has to switch between the threads. Consequently, for a four-core CPU with dual hyperthreading, the optimal number of threads would be between four and eight. Whether `hardware_concurrency()` returns 0, 4, 8, or something entirely different is not specified.

```
// https://godbolt.org/z/zojG68dWq
#include <thread>
#include <iostream>
#include <vector>
#include <chrono> // steady_clock
using std::cout; using std::endl; using namespace std::chrono;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }

int main() {
    cout << std::thread::hardware_concurrency() << '\n';
    for(int nthreads : { 1,2,3,4,5,6 }) {
        cout << "Threads: ";
        const auto start = steady_clock::now();

        std::vector<std::jthread> threads;
        for(int ti = 1; ti <= nthreads; ++ti) {
            threads.emplace_back( std::jthread{fib, 40});
            cout << ti << "... "; cout.flush();
        }
        for(auto &th : threads) th.join(); // explicitly join before timing

        const auto now = steady_clock::now();
        cout << "   Time: " << duration_cast<milliseconds>(
            now-start).count()<<"ms\n";
    }
}
```

Listing 29.11 Finding out the hardware concurrency.

Here, I sequentially calculate `fib(40)` one to six times in parallel and measure how long the calculations take. With `steady_clock::now()`, I get the current time before and after each series of calculations. The difference is the time taken. The `duration_cast` ensures that I get the output in a known time unit.

In the vector `threads`, I collect the thread instances. After starting the threads, I need to call `join()` on all elements of the container to wait until all are finished to measure the time.

For example, I have an Intel CPU i7-3520M with two cores, each with two hyperthreads. The previous program outputs 4 for me.

And the required times for parallel computation of `fib(40)` are as follows:

```
Threads: 1...    Time: 727ms
Threads: 1... 2...    Time: 758ms
Threads: 1... 2... 3...    Time: 1113ms
Threads: 1... 2... 3... 4...    Time: 1274ms
Threads: 1... 2... 3... 4... 5...    Time: 1703ms
Threads: 1... 2... 3... 4... 5... 6...    Time: 1955ms
```

Although 4 is returned for hardware parallelism, only two computations are actually fully parallelized. While the hardware may be able to execute up to four threads without significant context-switching overhead, they still steal computation time from each other. Note, however, that it was still faster to execute four `fib(40)` in parallel with 1274ms than to compute two `fib(40)` in parallel twice in a row, which takes $2 \times 758\text{ms} = 1516\text{ms}$.

29.1.8 Which Thread Am I?

The `std::this_thread` variable always represents the current thread. Its `get_id()` method returns a variable of type `std::thread::id`. You can compare this variable using `==` and `<`, and you can also apply `hash` to it. This means you can put it in associative containers like `map` and `unordered_set`. A thread can then identify itself and, for example, retrieve data associated with it from a container.

You can also output an `id`, but how the output looks is implementation-dependent.

```
// https://godbolt.org/z/TM6c7cdss
#include <thread>
#include <iostream>
int main() {
    std::cout << "Main: " << std::this_thread::get_id() << '\n';
    std::jthread th{ []{
        std::cout << "Thread: " << std::this_thread::get_id() << '\n';
    }};
}
```

Listing 29.12 Each thread has an identifier.

The output can be different with each program run, but it doesn't have to be:

```
Main: 139822070269824
Thread: 139822053463808
```

29.2 Shared Data

Each thread has its own stack. This means local variables and function parameters belong to it and only it. Another thread does not get to see them and, above all, cannot change them.

Global variables and heap memory are shared among all threads. If one thread changes a global variable, then another thread will notice—but *when* is hard to say.

Within a thread, it is guaranteed that one statement happens *after* the next. This may sound trivial, but it is not. I want to remind you of [Chapter 4](#), because what applies to statements does not apply to expressions: In `func(a(), b())` it is *not* specified that `a()` is executed before `b()` (or vice versa). Only from statement to statement is there a guarantee that operations between them appear to be completed. I emphasize the latter because the system promises to make it look to you as if the operations are completed. If the operations do not affect each other, the compiler or hardware is free to reorder them strategically. And not only that: Even if the statements refer to each other, they may be reordered if the same result is guaranteed. And believe me, current hardware and software make extensive use of these possibilities!

However, the guarantee only applies within a single thread! When multiple threads come into play, the well-intentioned reordering of instructions in one thread can lead to another thread, which relies on the sequence of instructions in the source code, encountering a completely different state. If it uses values based on its assumptions, it can be wrong, and the calculation yields an incorrect result.

The second point is that with true parallelism, instructions are indeed executed simultaneously. Even the simplest instructions like `i+=1` become problematic. Imagine that `i+=1` is executed exactly in parallel on two cores. Then `i` will not be incremented by 2, because for that to happen, each core would need to know at all times what each of the other cores is doing.

Therefore, when accessing shared data, it is important that it is *synchronized*:

- First, synchronization ensures that a computation in one thread is visible to another thread. In C++, you can use *atomics*—that is, atomic types.
- Second, you synchronize write accesses so that no other thread can write or read something incomplete at that moment. For this, you use mutexes and locks—for example, `mutex` and `lock_guard`.

29.2.1 Data Races

A *data race* occurs when a variable is simultaneously used (read or written) by multiple threads and at least one of them writes. The result of the operation is then undefined.

There is no data race if all involved threads only read.

```
// https://godbolt.org/z/7GEs9a4n6
#include <thread>
#include <iostream>

int count = 0; // is simultaneously modified

void run() {
    for(int i=0; i<1'000'000; ++i) {
        count += 1; // unprotected
    }
}

int main() {
    std::cout << "Start: " << count << '\n'; // Output: Start: 0
    std::thread th1{ run };
    std::thread th2{ run };
    std::thread th3{ run };
    th1.join(); th2.join(); th3.join();
    std::cout << "End: " << count << '\n'; // Output certainly not: 3000000
}
```

Listing 29.13 A classic data race.

A write operation `count+=1` in one thread may not be completed when another thread is also executing it. So it often happens that two threads increment the number, but only one of the results is written back to `count`, where it is then seen by the other threads. In my case, the result is not 3000000, but 1296367, and that is quite wrong! Oh, and it could have been something entirely different—up to a program crash, because the behavior is undefined.

In the context of C++, a data race usually refers to this “malignant” situation. Theoretically, “benign” data races are possible, but they do not exist in C++. The behavior of [Listing 29.14](#) is *undefined*. It may look like safe code and might even be safe on a given system, but generally, it is not.

```
// https://godbolt.org/z/v3b6sxEab
#include <thread>
/* exact count not so important */
int count = 0; // is concurrently modified
void run() {
    for(int i=0; i<1'000; ++i) {
        count += 1; // unprotected
        if(count > 1000) return; // ✎ Termination condition
        for(int j=0; j<1'000; ++j)
            ;
    }
}
```

```

    }
}

int main() {
    std::thread th1{ run };
    std::thread th2{ run };
    std::thread th3{ run };
    th1.join(); th2.join(); th3.join();
}

```

Listing 29.14 Benign data races are also undefined.

The intention here was probably to introduce a rough termination condition in the threads. And when the value `count` approximately reaches 1000, the threads should finish. Even if this works on a given system, it can do unpredictable things on another system or at another time.

In such a case, use one of the synchronization mechanisms, such as a `mutex` or an `atomic<int>`; see [Section 29.9](#).

The synchronization mechanisms provided to you by the standard library are as follows:

- **latch**

One-time use latches with a value that only let threads pass when the value is 0. From C++20.

- **barrier**

Reusable barriers that only let threads pass once a certain number of threads have arrived. The `barrier` then additionally triggers a signaling function. Also from C++20.

- **Futures**

These wait until a result is available.

- **Mutexes**

These lock a resource so that only one thread can use it.

- **Semaphores**

The general case of mutexes that allow multiple threads to pass simultaneously.

- **Atomics**

These are special types that you can use without mutexes. However, they can be unnecessarily slow.

I have sorted the tools by how complex they are to use correctly. You should therefore use `latch` or `barrier` if possible and only use `semaphores` and `atomics` if there is no other way.

29.2.2 Latch

You initialize the latch class with a value. Threads can then decrement the value. When the value reaches 0, all threads that were waiting for the value are restarted. This was added with C++20.

```
// https://godbolt.org/z/rrfn4d5WP
#include <thread>
#include <latch>
#include <iostream>
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }

int main() {
    std::latch la{ 3 }; // we expect 3 threads
    std::jthread th1{ [&la] { fib(39); la.count_down(); } };
    std::jthread th2{ [&la] { fib(38); la.count_down(); } };
    std::jthread th3{ [&la] { fib(40); la.count_down(); } };
    fib(37); // main thread
    std::cout << "Main thread: done\n";
    la.wait(); // waits until la == 0
    std::cout << "Rest done\n";
}
```

Listing 29.15 Latches can count down and wait.

Another method is `arrive_and_wait()`: if you call this first in a thread function, you can synchronize the start of multiple threads. This is useful, for example, for tests where, due to the small data size of the tests, the first thread might already be finished before the second one has started.

29.2.3 Barriers

The barrier class is a reusable barrier. You initialize it with the number of threads that should pass through it. When this number is reached, a signaling function that you can specify is triggered. The threads can then continue running. After that, the counter is reset to the original value. `barrier` was added with C++20.

The typical use case is the synchronization of parallel steps. The signaling function can, for example, pass the intermediate result of one step to the next step, write it to a file, and so on. The function runs within one of the involved threads.

```
// https://godbolt.org/z/515dMo9Kx
#include <thread>
#include <barrier>
#include <iostream>
#include <vector>
```

```

long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }

constexpr int anz = 8;           // 8 workers
constexpr int max_n = 32;        // calculate up to 32
std::vector<long> results(anz); // Buffer: space for 8 results

void output() {                  // Signaling function prints buffer
    for (auto n : results) std::cout << n << ' ';
    std::cout << '\n';
}

std::barrier ba{anz, output};    // always output after 8

void worker(std::stop_token st, int idx) {
    // n = 0, 9, 17, 25, ... ; 1, 10, 18, 26, ...
    for(int n = idx; n<max_n; n += anz) {
        if(st.stop_requested()) return;
        results[idx] = fib(n); // write result to buffer
        ba.arrive_and_wait();  // wait until 8 threads are here
    }
}

int main() {
    std::vector<std::jthread> threads; // 8 Threads
    for (int idx=0; idx<anz; ++idx) {
        threads.emplace_back(worker, idx); // create thread with index
    }
    for (auto& t : threads) t.join();    // wait until all are done
}

```

Listing 29.16 A barrier waits for a specific number of threads.

Each worker function calculates the Fibonacci numbers in steps of eight and writes the result to *its* position—so the threads do not interfere with each other. The function output outputs all results. And this happens exactly when eight threads in `worker` reach the call `arrive_and_wait()`.

The output is this:

```

0 1 1 2 3 5 8 13
21 34 55 89 144 233 377 610
987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229 832040 1346269

```

So 32 Fibonacci numbers are calculated, always with eight in parallel. The output always occurs when eight threads are done.

Other methods that might be useful are as follows:

- **arrive()**
Provides a token `t` for another thread, which can then call `wait(t)`.
- **wait()**
Waits for a token generated by `arrive()`.
- **arrive_and_wait()**
Is equivalent to `ba.wait(ba.arrive())`.
- **arrive_and_drop()**
Permanently decreases the counter but does not wait.

Futures

Although you should consider using futures first before considering mutexes, I describe the latter first because futures work with `async` instead of `thread` or `jthread`.

29.2.4 Mutexes

Mutex stands for *mutual exclusive*. You use a mutex on a resource that you want to protect from concurrent access: one resource thus has one mutex, no matter how many threads you use. If you have two instances of the resource, you need two mutexes.²

In each thread, you lock the mutex associated with the resource before using the resource. If another thread then wants to use the same resource, it also tries to lock this mutex. This fails, and the second thread can either wait or forgo using the resource.

Once the first thread is done with its transactions, it unlocks the mutex. If another thread was waiting to lock the same mutex, it will acquire the lock at the next opportunity and can then perform its actions.

In this way, the threads work cooperatively, mutually excluding each other from the critical resource, coordinated via a mutex.

A Mutex Is a Special Case of a Semaphore

A *semaphore* allows a fixed number of threads to simultaneously use a resource. A mutex is a semaphore with a limit of one: at most one thread can use the resource at the same time.

In C++20, the `counting_semaphore` and `binary_semaphore` classes have been newly added. They are, in a way, lighter alternatives to `mutex`. You do not use them with a lock (i.e., without `RAII`), but instead call `acquire` and `release` manually. The other difference

² This is a simplified view. Of course, there can be situations where you want to protect a resource from different things using multiple mutexes.

is that it does not matter which thread releases the semaphore, whereas with `mutex` it must always be the original thread.

What does this look like in C++ practice? You use an instance of the `mutex` class belonging to the resource and a `lock_guard` on the `mutex` in one thread to lock and unlock it.

With the `lock_guard` class, you cannot forget to unlock the `mutex` again, because the destructor takes care of that, following the RAIL principle. Thus, the range between initializing the `lock_guard` instance and its scope end is protected.

For special tasks, there are suitable alternatives to both `mutex` and `lock_guard`, which I will discuss later.

The following listing shows first a typical, albeit simple, use of the two basic tools.

```
// https://godbolt.org/z/3fz5a4vYv
#include <mutex> // mutex, lock_guard
#include <list>
#include <algorithm> // find
using std::lock_guard; using std::mutex; using std::find;
class MxIntList {
    std::list<int> data_;
    mutable mutex mx_;
public:
    void add(int value) {
        lock_guard guard{mx_}; // protects until the end of the method
        data_.push_back(value);
    }
    bool contains(int searchVal) const {
        lock_guard guard{mx_}; // protects until the end of the method
        return find(data_.begin(), data_.end(), searchVal) != data_.end();
    }
};
```

Listing 29.17 A `mutex` together with a simple lock.

Because the resource `data_` and its `mutex` `mx_` are closely related, I have encapsulated them together in a class here. This is particularly useful because you need to protect *every* access to `data_`. And in this way, you prevent users from accidentally using the list without a `mutex`. You don't have to put `data_` and `mx_` together in a class, but I strongly recommend it.

Critical Section

The code block protected by the locked `mutex` is called a *critical section*.

The use of `lock_guard` in `add` and `again` in `contains` means that accesses within these methods are mutually exclusive: `contains()` will never see a list that is being modified within `add()`.

Why is this important? The `list` class manages two references per element: one to the previous element and one to the next. To add an element, the following must happen:

- The new value must be wrapped in an element.
- The current last element must point forward to the new element.
- The new element must point forward to `end()`.
- The new element must point backward to the current last element.

There can be a moment when it would be catastrophic if an inserting thread is interrupted during its work between these steps and another thread wants to insert. Interruption is one thing, but with multithreading, as explained at the beginning, there is also the possibility of simultaneous events where the results are not visible to all threads. Even just writing the forward reference of the current last element can yield unexpected results if executed simultaneously by two threads.

Mutable Mutex?

A method like `contains()` should be `const`; after all, `data_` is only being read. However, as `mx_` is also a data field of the class, it should not be modified either. Locking the mutex changes it, however. Together with `mutable`, it is still possible to keep the `const` method.

Here I show you one of the rare meaningful uses of `mutable` data fields. You should not have to soften your interface just because you choose a particular form of implementation. Externally, it should not be visible to users that you are synchronizing threads with an internal data field.

Not quite as critical, but still not safe, is when another thread wants to read the end of the list just as an item is being added.

Protected like this, you can use `MxIntList` with multiple threads without any issues.

29.2.5 Interface Design for Multithreading

I have a simple stack class here, meaning I want to add and remove elements from the top and know if the stack still contains elements. In [Listing 29.18](#), I show you the public part and method declarations, not their definitions. You should notice that this class will have difficulties when used in parallel and is not inherently safe. The problem here is that you cannot rely on the result of `isEmpty()` in parallel operation. Although the

function safely returns a correct result at the moment of the call, as soon as it exits, other threads are free to call `push()` or `pop()`.

// <https://godbolt.org/z/4j7njd763>

```
#include <vector>

template<class T>
class MxStack {
public:
    bool isEmpty() const;
    void push(const T&);
    void pop();
    const T& top() const;
};
```

Listing 29.18 The interface to a multithreaded stack.

If the thread that evaluates `isEmpty()` acts on its result, it may be that this is no longer correct. The same is true for `top()` and `pop()`: the element that `top()` returns does not necessarily have to be the same one that `pop()` removes in parallel operation.

Consider the following listing.

```
// https://godbolt.org/z/1boeTMMvf
MxStack<int> mxs{};
// ...
// more code
// ...
if( ! mxs.isEmpty() ) {                // ⚡ not safe
    const auto value = mxs.top();      // ⚡ not safe
    mxs.pop();                         // ⚡ not safe
    // ...
    // more code
    // ...
}
```

Listing 29.19 Problematic code for the “MxStack”.

There are two possible data races here, the first between the calls to `isEmpty()` and `mx.top()`. Assume a stack `mxs` holds an element `{3}` and a thread reaches `!isEmpty()`. Because `!isEmpty` is true, the then branch is entered. But at this moment, another thread is executing the preceding code and performs `mx.pop()`—and the last element is removed! Continuing with the first thread, it now executes `top()` on the empty stack in the `if`. This suddenly doesn't work anymore, even though `!isEmpty()` just assured that the container was not empty. This is a nasty trap.

The second data race is a bit trickier to detect; it is between `top()` and `pop`.³ I will make the execution of two threads from the previous code a bit clearer. The timeline runs from top to bottom, with the threads displayed side by side.

```
/* Thread 1 */                                /* Thread 2 */
if( ! mxs.isEmpty()) {                        if( ! mxs.isEmpty()) {
    const auto value = mxs.top();              const auto value = mxs.top();
    mxs.pop();                                mxs.pop();
    // ... more code ...                      // ... more code ...
}
```

Listing 29.20 A possible way two threads could execute the previous code.

Assume we started with a stack `mxs = {1, 2, 3}`. Then in the first two lines, the threads each correctly determine that the stack is not empty. However, both retrieve the top element 3 from the stack using `top()`. Only then do both stacks execute `pop()` and remove not only the 3 but also the 2. The result is that the 3 is processed twice, while the 2 is not considered at all.

Even if I have internally protected `MxStack` very well with mutexes against inconsistent operations, the overall use of the `MxStack` interface is still inadequate for parallel operation.

The solution lies in changing the interface: it must ensure that typical operations are consistent. What that means depends heavily on the application. When in doubt, you should offer a lean interface that is not overloaded with unnecessary methods. This makes it easier for users to recognize the correct operation from the interface specification.

An example that solves our problem consists of changes in the interface:

- `top()` and `pop()` must be combined into a single call.
- `isEmpty()` alone is unsafe and must be complemented by an exception that can be thrown by `pop()`.

A simple interface with implementation could look like the following listing.

```
// https://godbolt.org/z/vcb9qP8jK
#include <vector>
#include <thread>
#include <mutex>
#include <iostream>
```

³ Strictly speaking, this is not a data race with undefined behavior, but rather a race condition, which leads to hard-to-find misbehavior.

```

#include <numeric> //iota
#include <concepts> //copyable

/* T: noexcept copyable and assignable */
template<std::copyable T>
class MxStack {
    std::vector<T> data_;
    std::mutex mx_;

public:
    MxStack() : data_{} {}

    bool isEmpty() const { return data_.empty(); }

    void push(const T& val) {
        std::lock_guard<std::mutex> g{mx_};
        data_.push_back(val);
    }

    T pop() {
        std::lock_guard g{mx_};
        if(data_.empty())
            throw std::length_error{"empty stack"};
        T tmp{std::move(data_.back())};
        data_.pop_back();
        return tmp;
    }
};

int main() {
    // Prepare stack
    MxStack<int> mxs{};
    for(int i=1; i<=1'000'000; ++i) mxs.push(i);
    // Define computation
    auto sumIt = [&mxs](long &sum) {
        int val{};
        try {
            while( ! mxs.isEmpty()) {
                sum += mxs.pop(); // might still throw
            }
        } catch(std::length_error &ex) {}
    };
    // Compute
    long sum1 = 0;           // for partial result

```

```

std::jthread th1{sumIt, std::ref(sum1)};
long sum2 = 0;           // for partial result
std::thread th2{sumIt, std::ref(sum2)};
th1.join(); th2.join();
long sum = sum1 + sum2; // Total result
// Result
std::cout << "Expected result: "
            << (1'000'000L*1'000'001)/2 << '\n'; // Output: 500000500000
std::cout << "Actual: "
            << sum << '\n';                      // Output: 500000500000
}

```

Listing 29.21 A very simple thread-safe stack.

Regarding concurrency, the following can be said:

- Both push and pop are each protected with a `lock_guard` on the same mutex, so they do not interfere with each other.
- `isEmpty` does not need to be protected.
- `pop` can throw an exception if the container is empty.
- `pop` attempts to move the element out of the internal vector. For this, `T` should support moving. If not, then it will be copied.
- It is important that copying or moving `T` does not throw an exception. If you do this during `pop()`, the value is not successfully written but already removed from the stack. This is called *not exception-safe*.
- To make the `pop` method exception-safe, you could take other measures, but that goes beyond the scope of this chapter.

Outside the influence of `MxStack`, I had to consider one more thing for parallelism. I let `th1` and `th2` add their sums in separate variables, `sum1` and `sum2`. If I did this in a single `sum` variable, I would have to protect the write access `sum += val` with its own mutex again.

As you know, `sum1` and `sum2` would be copied into the thread, even though the parameter of `sumIt` is a reference parameter. Therefore, I wrap the variables in `std::ref` when passing them to the thread to maintain the reference property.

Consider for Each Mutex Whether You Really Need It

Mutexes or the locks on them significantly slow down the computer. If you find a way to perform operations without mutexes, it is worth doing so.

Sometimes it is also advantageous to copy parts of the input into each thread beforehand or—as in the example—to merge partial results at the end.

First, you are likely to speed up the program by avoiding locks, and second, you have one less potential source of error—namely, shared data access.

A general remark is that copying and assignment of `MxStack` are automatically prohibited because the data field `mutex` is neither copyable nor assignable. The interface of `MxStack` could make this clear by listing both operations again with `= delete`. Alternatively, you can also implement the operations. However, I would only do this if it is really necessary, as it is not easy to implement perfect copy and assignment operations by hand. If necessary, I would rather recommend the move operations. On the one hand, it helps that you can move `mutex`, and on the other hand, you can then return `MxStack` from a function, for example.

29.2.6 Locks Can Lead to Deadlock

Multiple locks on a mutex can potentially lead to a *deadlock*. For example, if a method of `MxStack` calls another method and both try to lock the mutex, the second call will wait forever. In pseudocode, it looks something like this:

```
class MxStack {
    bool isEmpty() const {
        std::lock_guard<std::mutex> g{mx_};
        return data_.empty();
    }
    T pop() {
        std::lock_guard<std::mutex> g{mx_};
        if(data_.isEmpty()) ... // ✗ lock inside a lock
    }
};
```

A similar situation can arise when two operations `a` and `b` in two different threads need to lock two mutexes `mx` and `my`. Once again in pseudocode:

Thread 1: operation <code>a()</code>: <code>mx.lock()</code> <code>my.lock()</code> /* Work a... */	Thread 2: operation <code>b()</code>: <code>my.lock()</code> <code>mx.lock()</code> /* Work b... */
--	--

Under the circumstances presented here, Thread 1 waits for `my` to be released, and Thread 2 waits for `mx` to be released. But neither will ever happen because each thread is waiting—a deadlock.

Always Lock Multiple Mutexes in the Same Order

If multiple operations each need to lock multiple mutexes, it should always *happen in the same order*. At least for simple scenarios, a deadlock can then be ruled out. In more complex systems, this guaranteed same order is not always possible in practice.

The standard library offers a way to lock multiple mutexes simultaneously, which eliminates the risk of a deadlock for these scenarios. A typical example is the `swap` method of a large data structure, where each instance is protected by its own mutex, such as `MxStack` from [Listing 29.21](#). In the next listing, you see an example definition of a `swap` implementation.

```
// https://godbolt.org/z/9PMcz1cqP
friend void swap(MxStack& re, MxStack& li) {
    if(&re==&li) return; // Same address? Swapping with itself is unnecessary
    std::lock( re.mx_, li.mx_ ); // multiple locks simultaneously
    std::lock_guard lkre{re.mx_, std::adopt_lock}; // already locked
    std::lock_guard lkli{li.mx_, std::adopt_lock}; // already locked
    std::swap(li.data_, re.data_); // perform swap
}
```

Listing 29.22 “swap” for “MxStack”.

In `swap`, you need to lock the mutexes of both parameters to ensure that no other thread changes either parameter to an inconsistent state during the swap. Locking two mutexes means a risk of deadlock. The solution is this:

- First, lock both mutexes simultaneously with the function call `lock(...)`.
- To ensure that the locks are automatically released, use `lock_guard` again, but you must indicate with the additional parameter `adopt_lock` that the mutexes are already locked and do not need to be locked again in the `lock_guard` constructor.

As usual, `lkre` and `lkli` release their locks again when leaving the critical section.

Here are the general notes on the listing: I implemented `swap` according to the usual idiom—namely, as a `friend` function—so that it can be used as a free function like `swap(a,b)` instead of a method like `a.swap(b)`. This is the way the function best cooperates with the standard library. Within `swap`, you should first check, similar to assignment, whether `swap(a,a)` has been called directly or indirectly, making the swap unnecessary and perhaps even dangerous.

Tips for Avoiding Deadlocks

It is difficult to completely avoid potential deadlocks. Nevertheless, it is essential to achieve this. Therefore, careful consideration and testing are important—and even

more so because finding errors related to threads in general and deadlocks in particular is a very difficult task.

Therefore, here are a handful of general tips:

- **Avoid nested locks.**

If you are already holding a lock, you should not acquire another lock. This way, you could completely avoid lock-based deadlocks. In practice, you should do this as rarely as possible.

- **Do not call user code while holding a lock.**

This follows from the previous guideline, as you do not know what the user-provided code does. This is also not always feasible in practice.

- **Establish your locks in a fixed order.**

When you establish multiple locks on mutexes, strive to always do so in the same order.

- **Define a hierarchy of locks.**

If a clear, fixed order of locks is difficult to maintain, define a hierarchy that you or future users of the mutexes should adhere to. A high-level function should always lock only the high-level mutex and call low-level functions. Low-level functions should only lock low-level mutexes.

- **Consider deadlocks even in places that do not directly involve attempting to lock a mutex.**

Whenever it comes to waiting, you can run into a deadlock: `join()`, mutex locks, reading data from a file that is being written elsewhere, and many other things can interfere with each other.

29.2.7 More Flexible Locking with “unique_lock”

While `lock_guard` offers the simplest interface to use, you can alternatively use the more flexible `unique_lock`:

- `unique_lock lk{..., adopt_lock}`

The mutex is already locked; just take care of unlocking.

- `unique_lock lk{..., defer_lock}`

Do not lock the mutex immediately. Handle it later with `lk.lock()` (not on the mutex!) or with the `std::lock(lk)` function call.

- `unique_lock lk{..., try_to_lock}`

Try to lock, and if that doesn't work, proceed anyway.

Using `unique_lock` instead of `lock_guard`, the functionality would be identical. The big difference with `unique_lock`, however, is that it can temporarily *not own* the mutex associated with it. Similar to how a `unique_ptr` can transfer its content as a return value

or parameter to another `unique_ptr`, the `unique_lock` can do the same with the mutex associated with it. You can thus transfer the lock of a mutex out of or into a method or function.

```
// https://godbolt.org/z/e7rraPGx4
#include <thread>
#include <mutex>
#include <vector>
#include <numeric> // accumulate, iota
using std::mutex; using std::unique_lock;

std::vector<int> myData;           // shared data
mutex myMutex;                   // Mutex for the data
unique_lock<mutex> prepareData() {
    unique_lock lk1{myMutex};     // lock
    myData.resize(1000);
    std::iota(myData.begin(), myData.end(), 1); // 1..1000
    return lk1;                  // Transfer lock
}
int processData() {
    unique_lock lk2 = prepareData(); // Lock transferred
    return std::accumulate(myData.begin(), myData.end(), 0);
}
```

Listing 29.23 Mutexes can be transferred.

Here, the work on the `myData` data is distributed across two functions. First, `myMutex` is locked with `lk1`. So no one can pass a point that also wants to lock `myMutex`; this is nothing new. Under the protection of the lock, the `prepareData` function now manipulates `myData`, and no one can interfere. If `lk1` were just a `lock_guard`, the lock would be released upon exiting `prepareData()`, and others could then change `myData`. But I'm not done with the work on `myData` yet: I still want to perform `accumulate` in the caller `processData()`. An intermediate change to `myData` would be catastrophic.

So instead of a `lock_guard`, I use a `unique_lock` and return it from `prepareData()`. In the `processData()` caller, the lock is moved to `lk2`: it remains active and `myData` stays protected.

Only when the scope of `lk2` is exited does it release the lock. If `unique_lock` can do more than `lock_guard`, shouldn't you always use `unique_lock`? No, because `unique_lock` requires a little more management and is therefore slightly slower.

It's best to use `lock_guard` when you can, and `unique_lock` when you must. This also has the advantage of implicitly communicating your intentions for the lock.

29.3 Other Synchronization Options

There are special tasks for which a mutex with a lock is unsuitable. In this section, I will show you a few cases that you can solve using the tools of the standard library.

29.3.1 Call Only Once with “once_flag” and “call_once”

It often happens that you do not want to create a large object in advance just because it might be used later. You want to create the object only on first use—the so-called *lazy initialization*, or sometimes called *late initialization*. With only one thread, you could hold a global `unique_ptr` to the object and initialize it only on first use.

```
// https://godbolt.org/z/T7vehq58b
std::shared_ptr<BigData> bigData{};
BigData& getBigData() {
    if(!bigData) bigData.reset(new BigData{});
    return *bigData;
}
int useBigData() {
    auto bigData = getBigData();
    // bigData->...
}
```

Listing 29.24 Late initialization with a single thread.

This is problematic with multiple threads. If two threads call `getBigData()` simultaneously, `BigData` will be initialized twice. You could protect the entire function with a mutex, but there's a more effective way, as in the following listing.

```
// https://godbolt.org/z/3vcnzfqfM
#include <mutex> // once_flag, call_once
std::shared_ptr<BigData> bigData{};
std::once_flag bigDataInitFlag;
void initBigData() {
    bigData = std::make_shared<BigData>();
}
int useBigData() {
    std::call_once(bigDataInitFlag, initBigData);
    // bigData->...
}
```

Listing 29.25 Late initialization with multiple threads.

This way, `useBigData` can be called within threads, and the standard library guarantees that `initBigData` is called exactly once. You see that the function is passed here as a

parameter to `call_once`, so it's similar to `jthread{...}`. You are correct if you suspect that you can also use lambdas and function objects here.

This Is Just One Way of Late Initialization

Chapter 13, Listing 13.7 demonstrates another way of late initialization with the Meyer's Singleton pattern. The examples shown here with `shared_ptr` and `once_flag` are just more widely applicable.

You can also use `once_flag` in a class as a data field alongside the resource. Then both are encapsulated together in an object-oriented manner.

The scope of `once_flag` also extends beyond that of a mutex. You can also lazily initialize any resource within a single thread in this way.

```
// https://godbolt.org/z/TcsrefEPH
#include <mutex> // once_flag, call_once
#include <memory>
struct Connection {
    void csend(const char *data) {} // dummy
    const char* crecv() {} // dummy
};
class Sender {
    std::shared_ptr<Connection> conn_;
    std::once_flag connInitFlag_;
    void open() {
        conn_.reset( new Connection{} );
    }
public:
    void send(const char* data) {
        std::call_once(connInitFlag_, &Sender::open, this); // method pointer
        conn_->csend(data);
    }
    const char* recv() {
        std::call_once(connInitFlag_, [this] {this->open();} ); // lambda
        return conn_->crecv();
    }
};
```

Listing 29.26 Late initialization can also be useful within a thread.

At this point, it doesn't matter whether you call `send` or `recv` first, the initialization of `conn_` occurs exactly once using `open`—and if you don't call either, it doesn't happen at all.

Here, the function to be called is a method, which is why the cumbersome syntax & `Sender::open` must be passed for a *method pointer*, as I show in `send`. And because a method always takes the `this` pointer as an implicit first parameter, you pass it as an additional argument—just as you would pass additional parameters like `arg` for `func` with `jthread{func, arg...}`.

If this method syntax seems daunting, you can use a lambda. I demonstrated this in `recv`. You need to make `this` available as a capture variable in the lambda. Just `[] {open();}` wouldn't have been enough, because `this` is not visible in the lambda by default.

Both calling methods are equivalent. In the real world, you should not use different calls in associated `call_onces` as it only confuses your readers and is also prone to errors.

You can now use the same `Sender` instance in multiple threads without any issues (provided it is acceptable for multiple threads to send and receive data from a `Connection` simultaneously). Even in the presence of multiple threads, `conn_` is only initialized once. Thus, `call_once` combines the functionality of a mutex with that of a Boolean flag.

29.3.2 Locking with “recursive_mutex”

A thread that tries to lock a mutex that it has already locked will fail due to undefined behavior. In rare cases where you need to lock a mutex multiple times, use a `recursive_mutex`.

You can call the `recursive_mutex lock()` method multiple times within a thread. Another thread must wait until its `lock()` succeeds, which means you have to call `unlock()` as many times as you have used `lock()`.

You can call `lock()` and `unlock()` manually on the mutex, but you can also use `lock_guard` or other lock objects to help.

```
// https://godbolt.org/z/KPaPoh4Gn
#include <mutex> // recursive_mutex
#include <iostream>
struct MulDiv {
    std::recursive_mutex mx_;
    int value_;
    explicit MulDiv(int value) : value_{value} {}
    void mul(int x) {
        std::lock_guard lk1(mx_); // inner
        value_ *= x;
    }
}
```

```
void div(int x) {
    std::lock_guard lk2(mx_); // inner
    value_ /= x;
}
void muldiv(int x, int y){
    std::lock_guard lk3(mx_); // outer
    mul(x);
    div(y);
}
};
int main() {
    MulDiv m{42}; // 3*7*2*5
    m.muldiv(5, 15);
    std::cout << m.value_ << '\n'; // Output: 14
}
```

Listing 29.27 Precautions are necessary when a mutex needs to be locked multiple times.

Here, calling `muldiv` causes `mx_` to be locked with a simple `lock_guard` via `lk3`. From now on, no other thread can perform calculations on this instance `m`. Other threads will have to wait because each method immediately acquires a lock. The main thread then enters `mul`. There, a `lock_guard` via `lk1` locks the same mutex again in the same thread; with a simple mutex this would be an error, but with a `recursive_mutex` it is allowed. As soon as `mul` is exited, this inner lock is released again. However, this is not enough for other threads that might already be waiting to proceed. Only when the outer `lk3` releases its lock can the other threads continue.

If you are writing a program where you think a `recursive_mutex` is the solution, it is usually an indication that you should reconsider your design. In a class, a mutex typically protects the encapsulated data. You need a recursive mutex if a public method calls another public method. However, a lock usually exists because the “invariant” of the class is violated in the *critical section*. Does it sound like a good idea to call a public method while the instance is in an incorrect state? No. It is often better to define a private method that is then called by both public methods. You usually establish the lock at the public level, and the private method can assume that a lock already exists.

29.4 In Its Own Storage with “`thread_local`”

Not all memory is visible to all threads. If you define a variable with `thread_local` in a running thread, only the thread that created it can see it. When the thread terminates, the variable is also removed.

```
// https://godbolt.org/z/1j8W7GjTf
#include <iostream>
```

```

#include <string>
#include <thread>
#include <mutex>
thread_local unsigned int usage = 0;
static std::mutex cout_mutex;
void use(const std::string thread_name) {
    ++usage;
    std::lock_guard lock(cout_mutex); // Protect output
    std::cout << thread_name << ": " << usage << '\n';
}
int main() {
    std::jthread a{use, "a"}, b{use, "b"};
    use("main");
}

```

Listing 29.28 Memory only for the current thread.

You will get the following output (in this or another order):

```

main: 1
b: 1
a: 1

```

The 1 shows that each thread `a`, `b`, and `main` got its own `usage` variable and did not have to share it with other threads. Otherwise, a 3 would have been displayed at the end.

You use `thread_local` like `static` or `extern` and cannot combine it with these. Within a function, the variable is initialized once per thread, just as `static` would do with only one thread:

```

void func() {
    thread_local vector<int> data{};
    // ...
}

```

29.5 Waiting for Events with “condition_variable”

With mutexes and locks, you can go very far, and these form the basis of what you need to work with threads. Sometimes, however, handling mutexes and locks is cumbersome. With `call_once`, you have already seen an alternative way where the standard library assembles the basic building blocks into something new for you. There are a few more of these advanced tools that I will introduce to you here.

A typical scenario with multiple threads is that one produces data and the other waits for it to be ready for processing. These two can communicate with each other via a condition variable `condition_variable`.


```
// https://godbolt.org/z/c9x6xT8dr
#include <thread>
#include <mutex>
#include <condition_variable>
#include <vector>
#include <deque>
#include <iostream>
#include <syncstream> // osyncstream
std::deque<int> g_data{}; // Data exchange between threads
std::condition_variable g_condvar; // notify
std::mutex g_mx; // protects g_data during changes
void produce(int limit) {
    std::vector primes{2}; // prior primes as test-divisors
    for(int candidate=3; candidate < limit; candidate+=2) {
        for(int divisor : primes) {
            if(divisor*divisor > candidate) { // candidate is prime
                std::lock_guard lk{g_mx}; // protect data
                g_data.push_back(candidate); // fill
                g_condvar.notify_one(); // notify
                primes.push_back(candidate); // for internal calculations
                break; // next prime candidate
            } else if(candidate % divisor == 0) { // not prime
                break; // next prime candidate
            } else {
                // next divisor to check
            }
        }
    }
    // notify all work done
    std::lock_guard lk{g_mx}; // protect data
    g_data.push_back(0); // fill with end marker
    g_condvar.notify_all(); // notify
}
void consume(char l, char r) {
    while(true) { // forever
        std::unique_lock lk{g_mx};
        g_condvar.wait(lk, []{ return !g_data.empty();});
        int prim = g_data.front(); // fetch data
        if(prim == 0) return; // done; leave 0 for other consumers
        g_data.pop_front();
        lk.unlock(); // release lock
        std::osyncstream osync{std::cout}; // synchronize output
        osync << l << prim << r << ' ';
    }
}
```

```

int main() {
    // a producer:
    std::jthread thProd{produce, 1'000};
    // three consumers
    std::jthread thCon1{consume, '[', ']'};
    std::jthread thCon2{consume, '<', '>'};
    std::jthread thCon3{consume, '{', '}' };
    // wait and finish
    thProd.join();
    thCon1.join(); thCon2.join(); thCon3.join();
    std::cout << '\n';
}

```

Listing 29.29 Two threads communicate via a condition variable.

Here, there is a producing thread `thProd` in the function `produce` and three consuming threads `thCon1` to `thCon3` in the function `consume`. The latter should output the data. To distinguish which thread handles the output, I use different pairs of brackets, which I pass as parameters to the `consume` function.

The produced data is placed in the global queue `g_data` as quickly as the producer can deliver it. So it may be that `g_data` fills up faster than results can be consumed, but it may also be that the consumers empty `g_data` and really have to wait for new results. In the case of prime number calculation, it initially runs quickly and then gradually slows down. However, 1'000 or even 1'000'000 is far too small a number to notice on modern computers. To output fewer numbers, you can add another test to check if a prime sextuplet is present.⁴

Ignoring the specific computational task, you need to consider the following for parallel programming here:

- Data is exchanged via `g_data`. Changes to this data structure must always be protected. Here I use the mutex `mx` and the corresponding locks for this purpose.
- Whenever data is ready in `g_data`, the producer notifies any waiting threads with `notify_one` that work is ready for pickup.
- On the consuming side, the first consumer locks the mutex `mx`, but this time with `unique_lock lk`. This is important for working with `wait`.
- I then pass the lock `lk` along with a test to the `wait` method of the `condition_variable` to check if there is actually work to be done. Here I test in the lambda if `g_data` actually contains data.
- If the lambda of `wait` returns `false`, `wait` releases the lock `lk` (hence `unique_lock`, because `lock_guard` does not have an `unlock()` method), so that a producer can

⁴ See https://en.wikipedia.org/wiki/Prime_tuple.

actually write data to `g_data` again. Then the `wait` method blocks and puts the thread into a sleep state to minimize system load.

- As soon as the producer then calls `notify_one()` on `g_condvar`, a consumer slumbering via `wait()` is awakened.
- The awakened consumer first tries to lock `lk`. Once it succeeds, `wait()` rechecks the lambda condition.
- If the lambda returns `true`, `wait` returns to its caller but keeps `lk` locked.
- The caller, here `consume()`, can then continue with its work. First, it should use the data structure protected by `lk` so that it can release the lock as quickly as possible. Here, I do this manually with `lk.unlock()`. However, if the scope of `lk` is exited differently, `unique_lock` will automatically handle the unlocking as usual.

In the consumer, as always, as little time as possible should be spent under the lock. Often, the work that the consumer performs is inherently time-consuming. Maintaining the lock throughout the entire output would be a huge waste of time. Therefore, it is quite common to quickly get a local copy of the data from the shared data structure, here `g_data`, release the lock, and then work on the local data copy at leisure.

In this specific example, it is indeed the case that prime number calculation is many times faster than outputting to `cout`. Even when calculating up to 1 million, three output threads cannot keep up.

29.5.1 “notify_all”

Finally, the producer must inform the consumer that all the work is done and completed. He can do this in various ways, such as via another `condition_variable`. In this specific example, this is not necessary because I simply use the 0 as an end marker in `g_data`:

```
g_data.push_back(0);
```

When a consumer reads a 0, they know that the computation is finished, and the thread function can terminate with `return`:

```
int prim = g_data.front();  
if(prim == 0) return;
```

It is important not to execute `g_data.pop_front()`; otherwise the end marker would no longer be there for other consumers.

However, I want to inform not just one consumer about the presence of the end marker, but all of them. Therefore, instead of using `notify_one()`, I use the `notify_all()` method.

Advantages of the Condition Variable

The `condition_variable` is designed to hold the necessary locks for as short a time as possible. Thus, during a `wait()` call, the conditions of the lambda may be checked multiple times. For this, the mutex is locked but always immediately unlocked again to allow others to work on the shared data structure.

Rebuilding this by hand is laborious, error-prone, or very likely results in suboptimal code. You should use a `condition_variable`.

29.5.2 Synchronize Output

The program output looks something like this if you choose 1'000 as the limit:

```
[3] <5> [11] {7} [13] [19] {23} [29] <17> [31] {37} <41> <47> [43]
{53} <59> {67} [61] <71> {73} [79] <83> {89} [97] <101> {103} [107]
<109> ... {881} [887] [919] <907> {911} [929] {941} [947] <937> [953]
{967} <971> [977] [997] {983} <991>
```

You can tell from the different brackets that the numbers are indeed output by different threads. You can also see that the numbers do not always appear in the correct order. This is an intentional design decision. Often, you want the data to be consumed as quickly as possible. Who consumes it and in what order relative to each other is often irrelevant. Maybe they are independent work orders from the internet. However, if the output order is important, use a different mechanism.

For the output, you see that the output does not happen directly after `cout`, but with an instance `osync` of type `osyncstream`:

```
std::osyncstream osync{std::cout}; // Synchronize output
osync << l << prim << r << ' ';
```

If you omit this, the streams could interrupt each other during output, and the output would be a jumbled mess (to be precise, *undefined*). The `osyncstream` class works similarly to a `unique_lock`, except you don't need an additional mutex.

Don't have a `osyncstream` available yet? Then define a mutex globally and use a `unique_lock`. Alternatively, you can also collect the output that should be kept together in a `stringstream` first, so that you hold the lock for as short a time as possible:

```
std::mutex g_coutMx;
// ...
std::stringstream ss;
ss << l << prim << r << ' ' ;
std::lock_guard<std::mutex> lk2{g_coutMx};
std::cout << ss.str();
```

29.6 Waiting Once with “future”

Getting a result back from a thread is not so easy with what you know so far. But fortunately, there is more in the standard library.

For example, it is often the case that you want a thread to perform a calculation in the background, but in the main thread, you want to perform other tasks in parallel before retrieving the result of the background calculation. In that case, you use `async` to start the thread. This function returns a `future`, which you can later query to see if the result is already there, or you can simply wait until the result is ready.

Unlike `condition_variable`, `future` is meant for situations where you want to receive a result only *once*. The practical thing about `future` is that it packages the return value of the thread directly into `future`, so you don't have to manage it in a separate data structure. Here `future` is a template: `future<int>` delivers an `int` result, similar to how `unique_ptr` packages an `int` value. There is also `shared_future`, which allows multiple threads to wait for the result.

Like `jthread`, `async` has easy-to-use exception handling—and unlike `thread`. I will go into detail on this later, but for now, just know this: An exception thrown by the thread function is stored temporarily and presented to the retrieving thread as an alternative result. This greatly simplifies centralized exception handling.

Enough theory, let's just try it out.

```
// https://godbolt.org/z/Gh9Y5srd7
#include <iostream>
#include <future> // async
using std::cout; using std::endl;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }

int main() {
    auto f40 = std::async(fib, 40);
    auto f41 = std::async(fib, 41);
    auto f42 = std::async(fib, 42);
    auto f43 = std::async(fib, 43);
    /* ... at this point, further calculations can be made ... */
    cout << "fib(40): " << f40.get() << endl; // Output: fib(40): 102334155
    cout << "fib(41): " << f41.get() << endl; // Output: fib(41): 165580141
    cout << "fib(42): " << f42.get() << endl; // Output: fib(42): 267914296
    cout << "fib(43): " << f43.get() << endl; // Output: fib(43): 433494437
}
```

Listing 29.30 “`async`” can easily initiate asynchronous computations.

As you can see, you start a thread using `async` in the same way you would with `jthread`: the first argument is a callable object (function pointer, lambda, functor, etc.), and the rest are parameters that are passed as copies to the thread function at startup.

The difference is that here you do not receive a `jthread` or a plain thread as a return, but a `future<...>`, specifically a `future<long>`. Because there is rarely a reason to write out this type on the left side, you will mostly see `auto` there, as shown here.

Once you have started all the desired threads, the starting thread can occupy itself with other tasks. Later, when the calling thread is ready, `get()` is the simplest way to obtain the result. The return type of `get()` is that of the thread function—in this case, `long`.

You don't need an explicit `join()`. Why would you? You don't have a `thread` or `jthread` object. This role is taken over by `get`. Unlike with `thread`, it does no harm if the scope of `future` is exited before the result is retrieved. Here, it is the destructor that waits until the computation is finally completed, similar to `jthread`.

```
// https://godbolt.org/z/7Mea1Y3Mn
#include <iostream>
#include <future> // async
using std::cout; using std::endl;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }

int main() {
    auto f40 = std::async(fib, 40);
    auto f41 = std::async(fib, 41);
    auto f42 = std::async(fib, 42);
    auto f43 = std::async(fib, 43);
    cout << "fib(40): " << f40.get() << endl; // Output: fib(40): 102334155
} // also waits for f41, f42, and f43.
```

Listing 29.31 It is okay not to retrieve a result.

29.6.1 Launch Policies

By using `get`, it should feel to you as if the value has always been there for you and has been ready for pickup forever, very similar to a `unique_ptr`. If that is the case, then it doesn't really matter whether the computation took place in the background and you just need to pick it up with `get`, or if nothing has happened yet and the computation only starts with `get`. You wouldn't notice the difference from the outside (except perhaps in the elapsed time).

And indeed, `async` automatically chooses one of the two options for you. If you want to ensure that the computation is executed in one way or another, you can specify this as a rule in an extra parameter. The standard library calls this a *launch policy*.

There are exactly two launch policies of type `std::launch`:

- **`launch::async`**
Start a separate thread for the background computation.
- **`launch::deferred`**
Wait with the computation until the result is retrieved with `get`. This does not necessarily have to happen in a separate thread.

`launch` is an enum, whose values you can combine with the `|` operator (for *or*) if you want to leave the choice to the standard library. `launch::deferred | launch::async` is the default behavior.

An example of deferred execution looks like the following listing.

```
// https://godbolt.org/z/P9MjGe6Mn
#include <iostream>
#include <future> // async
#include <vector>
using std::cout; using std::endl;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
int main() {
    // Prepare tasks
    std::vector< std::future<long> > fibs;
    for(int n=0; n<50; ++n) {
        auto fut = std::async(std::launch::deferred, fib, n);
        fibs.push_back( std::move(fut) );
    }
    // only retrieve the required result
    cout << "fib(42): " << fibs[42].get() << endl; // Output: fib(42): 267914296
}
```

Listing 29.32 How to enforce deferred execution.

By the way, you don't have to wait for the result with `get` and retrieve it at the same time. You can also just wait first with `wait()`. A subsequent `get()` will then deliver the result immediately.

29.6.2 Wait Until a Certain Time

Instead of using `wait`, you can also wait for a certain time with `wait_for` or `wait_until`:

- **`wait()`**
Waits until the result is ready.
- **`wait_for(std::chrono::duration)`**
Waits until the result is ready or a duration has elapsed.

- `wait_for(std::chrono::time_point)`

Waits until the result is ready or a time point is reached.

The last two do not return `void`, but a `future_status`, which you can use to check if the result is already there or if the timeout has occurred.

```
// https://godbolt.org/z/o9oYrvMcj
#include <iostream>
#include <future> // async
#include <chrono>
using std::cout; using std::endl; using namespace std::chrono;
long fib(long n) { return n<=1 ? n : fib(n-1)+fib(n-2); }
int main() {
    auto f43 = std::async(fib, 43);
    while(true) {
        auto ready = f43.wait_for(500ms);
        if(ready==std::future_status::timeout) {
            std::cout << "not yet..." << std::endl;
        } else {
            break;
        }
    }
    // pick up, is immediately there
    cout << "fib(43): " << f43.get() << endl; // Output: fib(43): 701408733
}
```

Listing 29.33 Waiting for a certain time with “async”.

The output for me is this:

```
not yet...
not yet...
not yet...
not yet...
not yet...
not yet...
fib(43): 433494437
```

So it was checked six times, and the result was not yet there. On the seventh time, the result was present. The following `get()` then immediately delivers the result.

Waiting Forever or For a Certain Time

If you block in the thread area of the standard library with a function like `wait()` and can wait forever for the function to return, there are always functions that allow you to

limit the waiting time. In the case of `future::wait()`, these are `wait_for` and `wait_until`. Among the mutexes, there is class `timed_mutex` with methods `try_lock_for` and `try_lock_until`. With `unique_lock`, you pass a duration or a time_point as an additional constructor parameter.

An example of using durations with mutexes and locks can be seen in [Listing 29.34](#). There, it is `try_lock_for` from `timed_mutex` that takes a duration as a parameter.

```
// https://godbolt.org/z/xzY8hj9oP
#include <chrono>
#include <future>
#include <mutex>
#include <vector>
#include <iostream>
std::timed_mutex mtx;
long fibX(long n) { return n < 2L ? 1L : fibX(n-1L) + fibX(n-2L); }
long fibCall(long n) {
    using namespace std::chrono; // Suffixes
    if(mtx.try_lock_for(1000ms)) {
        auto res = fibX(n);
        mtx.unlock();
        return res;
    } else {
        return 0L;
    }
}

int main() {
    std::vector< std::future<long> > fs;
    for(long n=1; n<= 42; ++n) {
        fs.emplace_back( std::async(std::launch::async, fibCall, n) );
    }
    for(auto &f : fs) {
        std::cout << f.get() << " ";
    }
    std::cout << std::endl;
}
```

Listing 29.34 Waiting for a lock with a “`timed_mutex`”.

The `fibX` function is supposed to simply “take a long time,” and the larger the argument `n` is, the longer it will take. Assume that `fibX` is not multithreading-capable: no two threads are allowed to enter it simultaneously. `fibCall` safely calls `fibX` and can also be called in parallel. With `try_lock_for(1000ms)`, the `fibX` call is protected from being

entered by two threads simultaneously. However, it waits for a maximum of one second. If it takes longer, `fibCall` returns zero instead of the result.

The output looks something like this:

```
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
10946 17711 28657 46368 0 0 0 317811 0 0 0 0 0 9227465 0 0 0 0 102334155 0 0 0
```

Whenever a call had to wait longer than a second, a zero appears. Initially, everything is fast enough, but later a call must be lucky to even be allowed to enter `fibX`.

The code in `main` is one way to have multiple functions executed in parallel. `async` immediately returns `future` results, which continue to be computed in the background. Only at the moment when the `get()` method is called on a `future` does the program wait for the computation.

29.6.3 Exception Handling with “future”

If the thread function started by `async` throws an exception, you must handle it at the place you call `get` to retrieve the result. Finally, with `launch::deferred` policy, it may be that the computation hasn't started yet and actually happens only at `get`.

Unlike with `thread`, you are not forced to handle exceptions *within* the thread function; you can do that in the calling thread. This increases the chance that you can keep exception handling centralized and avoid duplicating `catch` blocks.

But, you might ask, what happens if the exception is thrown in a thread started by `async` with the `launch::async` policy? The answer: it is caught and stored by the framework. It is thrown in the *calling* thread when you query the result with `get()`.

```
// https://godbolt.org/z/rYM9xqsGa
#include <future> // async
#include <vector>
#include <algorithm> // max
#include <iostream>

int calculateHeight(int count, int maxCount, int scale) {
    if(maxCount == 0)
        throw std::logic_error("All heights 0");
    return (count * scale) / maxCount;
}

void bar(const std::vector<int> &counts) {
    // Start calculation
    auto maxCount = *std::max_element(counts.begin(), counts.end());
    std::vector< std::future<int> > futs;
```

```
for(int count : counts) {
    futs.push_back(
        std::async(std::launch::async,
            calculateHeight, count, maxCount, 200) );
}

// Collect results
for(auto &fut : futs) {
    std::cout << fut.get() << ' ';           // triggers exception
}
std::cout << '\n';
}

int main() {
    try {
        bar(std::vector {10,23,13,0,33,4 }); // Output: 60 139 78 0 200 24
        bar(std::vector { 0, 0, 0, 0 });     // triggers exception
    } catch(std::exception &ex) {
        std::cout << "Error: " << ex.what() << '\n'; // Output: Error: All heights 0
    }
}
```

Listing 29.35 Exceptions only arrive at “get” in the outer thread.

The exception is triggered in `calculateHeight`—that is, in the thread function that was actually started in the background with `async` and the `launch::async` policy.

This program logic would not have been so simple with `thread`: First, sending a result to the calling thread would have required synchronization and some shared variable. And then to send a message somewhere deep in the calculation in case of an error? Feasible, but only with more code.

In the following listing, I have a central handling of all errors in `main`. It would also be sufficient to secure only the `get()` with `try`.

```
// https://godbolt.org/z/1WG5arvWT
#include <future> // async
#include <iostream>
using namespace std;
int calcHeight(int count, int maxCount, int scale) {
    if(maxCount == 0)
        throw logic_error("maxCount is 0");
    return (count * scale) / maxCount;
}

int main() {
```

```

auto fut = async(launch::async, calcHeight,0,0,200); // ⚡ throws
try {
    cout << fut.get() << '\n';           // triggers exception
} catch(exception &ex) {
    cout << "Error: " << ex.what() << '\n'; // Output: Error: maxCount is 0
}
}

```

Listing 29.36 Only the “get” needs to be encapsulated in “try”.

Even if `async` starts the calculation immediately and the exception occurs in the thread, it is initially stored. Only `get()` triggers the stored exception.

29.6.4 “promise”

A future can not only be created with `async`. Looking under its hood, you will find a *promise*—the means to pass the result or the exception from the executing thread to the caller. In this sense, you could see a promise as a kind of *channel* for communication between threads. This term also can be found in some other APIs.

The typical procedure when working with futures and promises is as follows:

- Create a new promise.
- Request a future associated with the promise.
- Start a new thread and pass it the promise.
- Perform the actual computation in the new thread.
- Assign the result to the promise.
- Wait in the caller until the future can deliver a result.
- Retrieve the result of the future.

Of course, the new thread in which the computation is performed asynchronously must have access to the promise in order to communicate with it. You can pass it to the new thread as an argument, for example. So a future and a promise always belong together.

The future remains with the caller to receive the message with `get()` (in channels, this roughly corresponds to *receive*). The called function gets the promise and uses `set_value` to send a result (which corresponds to *send* in channels). If an exception occurs, use the `set_exception` method instead.

Communication via “promise” and “future”

Specifically, this means that if you want to have a function asynchronously compute an `int`, you make a promise that you will compute the `int`. You create this with the following:

```
promise<int> intPromise;
```

From this promise, you then request a future with `get_future()` to keep in the caller, while you move `intPromise` itself into the asynchronous function with `move()`—you cannot copy a promise.

```
// https://godbolt.org/z/7E1E6aezE
#include <future>
#include <thread>
#include <iostream>
#include <exception>
int ack(int m, int n); // Ackermann function
void longComputation(std::promise<int> intPromise) {
    try {
        int result = ack(3,12);
        intPromise.set_value(result); // report result
    } catch (std::exception &e) {
        intPromise.set_exception(make_exception_ptr(e)); // report exception
    } catch ( ... ) {
        intPromise.set_exception(std::current_exception()); // Exc. w/o name
    }
}
int main () {
    std::promise<int> intPromise; // Create promise
    std::future<int> intFuture = intPromise.get_future(); // Request future
    std::jthread th{ longComputation, // start
                    std::move(intPromise) }; // Pass promise
    th.detach(); // let it run
    // might throw an exception:
    int result = intFuture.get(); // Request result
    std::cout << result << std::endl;
}
```

Listing 29.37 A “future” and “promise” working together.

In `main()`, the process described previously takes place. First, a new promise for an `int` result is created. Then, you request an associated future from it using `get_future`, which remains in the main thread.

In this example, I create a thread for the actual execution. But you can also come up with a more elaborate method to distribute the new task, such as an asynchronous queue that processes tasks for as long as there are any. The `intPromise` is passed as a parameter to the function to be executed in the thread; copying is not possible, so I move `intPromise` with `move()`. The subsequent `th.detach()` detaches the newly started thread from the variable `th`. I could have also ensured before exiting `main()` with

`th.join()` that `th` only becomes invalid when the thread is finished, but that is irrelevant for this example.

The function `longComputation` started by `th` received the `intPromise` as a parameter for communication with the caller. If the calculation leads to a regular result, you communicate this via `intPromise.set_value()`.

Because unhandled exceptions within `th` lead to `terminate()` and thus to the end of the program, your own exception handling is important here. If you want to communicate an exception, use `intPromise.set_exception()`. This is usually done in a `catch` block. If you hold the exception in a variable `e`, you can get a reference with `make_exception_ptr(e)` that the promise can store until it is retrieved with `get()`. If you do *not* hold the exception in a variable, you can obtain this reference with `current_exception()`.

In the main thread, the result is then requested with `get()`. This call waits until it is ready. This can be the regular value, which is then assigned to `result`, or a stored exception, which is thrown here in the main thread.

A combination of threads, promises, and self-created futures can, for example, realize *asynchronous input and output*.

Bundling Tasks Together

When you use `async`, the promise part of the communication conveniently does not appear as it would normally always look very similar anyway. However, the execution is either started *immediately* or at the moment of the `get()` call.

If you want more control over when the computation is executed, you can use `packaged_task`. Here too, users only deal with the future part of the communication. Preparing the computation function and triggering the computation are separated from each other.

```
// https://godbolt.org/z/9s94nbrx3
#include <future>
#include <thread>
#include <iostream>
int ack(int m, int n); // Ackermann function
int main () {
    std::packaged_task<int(void)> task1 { // Signature of the remaining function
        [{ return ack(3,11);} ];        // prepare ack(3,11)
    };
    auto f1 = task1.get_future();        // Communication channel
    std::jthread th1 { move(task1) };    // in new thread
    std::cout << "  ack(3,11):" << f1.get() // Retrieve result
               << '\n';                  // Output: ack(3,11):16381
}
```

Listing 29.38 Preparing a packaged task for later execution.

You are already familiar with `get_future` and `get`. What is new is that with `task1` you can already prepare the later function call. The template parameter of `packaged_task` reflects the signature of the function to be called later: `int(void)` is a function without parameters that returns an `int`. Here, it is specifically a lambda that will later perform the time-intensive computation.

Now you can trigger the execution of the function anytime with `task1()`; In the example, however, this should happen in a new thread. For this, you create a `jthread` object with the task as a parameter. Tasks cannot be copied, only moved, which is why `move` is necessary.

The task ensures that the calculation result is communicated internally via a promise and can be queried via the future `f1`. The `join()` ensures that the thread is cleanly terminated when `th1` goes out of scope.

For another scenario where you do not want to set all arguments at the moment of creating the `packaged_task` but only when starting the thread, there are small differences, as in the next listing.

```
// https://godbolt.org/z/bq1hn5rjo
#include <future>
#include <thread>
#include <iostream>
int ack(int m, int n); // Ackermann function
int main () {
    std::packaged_task<int(int,int)> task2 { &ack }; // different signature
    auto f2 = task2.get_future();
    std::jthread th2 { move(task2), 3, 12 };          // Parameters here
    std::cout << "  ack(3,12):" << f2.get() << '\n'; // Output: ack(3,12):32765
    th2.join();
}
```

Listing 29.39 Provide a packaged task with arguments later.

The signature that must be specified when constructing `packaged_task` is always the one that remains for the actual call—here through `jthread`. Because I pack `ack` into `task2` without already bound parameters, this is a function that takes two `int` and returns one—that is, `int(int,int)`. This signature determines whether I could call, for example, `task2(3,12)`; at any time. However, as this task is now to be taken over by `jthread`, these two parameters are now also specified during thread creation.

29.7 Atomics

I would like to give some advice here in the form of light warnings for dealing with an area of C++. Here are the most important guidelines for dealing with `atomic`:

- Do not use `atomic` unless you have to.
- Check the success through profiling before and after. Too often, a brilliant “optimization” turns out to be counterproductive.
- Stick to the default `memory_order_seq_cst` for the operation mode of `atomic`. The other variants only have more runtime benefits than implementation costs in the most extreme situations.

I recommend that you start with a variable protected by a mutex, gather experience, and then—maybe—experiment with `atomic`.

With that in mind, let me explain what you need `atomic` for.

You have seen that you can synchronize shared data between threads by locking. Well, locks have a side effect: they lock. That means it can happen that another thread has to wait at a lock, because that's exactly what they are made for. This could be considered suboptimal. Where there is waiting, time is lost. Yes, maybe a little, but don't forget that waiting also frees up resources that another thread can use. The actual loss is small, but it is there. In addition, there are application areas where *every* waiting can be problematic. For these cases, you need an alternative.

It is possible to achieve one of the following stages of the “guaranteed completion” process, which are referred to as *lock-free guarantees*:

- **Obstruction-free, the weakest guarantee**

If a single thread runs in isolation (without any other threads interfering), it will complete its operation in a finite number of steps. No guarantees are made if there is contention from other threads.

- **Lock-free**

This ensures that at least one thread will make progress in a finite number of steps, even if multiple threads are competing. This means that at least overall progress is achieved.

- **Wait-free, the strongest guarantee**

Guarantees that every thread will complete its operation in a finite number of steps, regardless of the actions of other threads, thus each thread achieves progress at any time.

You achieve obstruction-free with mutexes, and that is already a success. Lock-free is still an achievable goal if you are willing to put in a lot of effort. This is where `atomic` comes into play. But beware: the use of `atomic` is not a guarantee for a lock-free program, nor does a lock-free program necessarily require `atomic`. Sometimes you can achieve a lot with surprisingly simple tricks (perhaps a copy of the data?). Achieving the highest guarantee would of course be great, but it is incredibly difficult and only possible with a precise understanding of the system and the possible involved data down to the smallest detail. Here, `atomic` would also come into play, and you would probably even have to resort to the tricks of `memory_order` possibilities.

The name `atomic` refers to the *temporal indivisibility* of an operation. An atomic operation in one thread cannot become *partially* visible in another thread. The operation either has not started yet or is finished.

`atomic` is a template class. This means you provide it with a type as a parameter, so `atomic<T>`. In principle, this type can be anything, even one defined by you. In practice, the `T` falls into one of the following categories, which influence the behavior of `atomic<T>`:

- **`atomic<B>`, where `B` is a built-in type**
Supports all atomic operations; however, the standard does *not* guarantee that these are lock-free.
- **`atomic<U>`, where `U` is a user-defined type**
`U` must be copyable with `memcpy` (trivially copyable); better yet, comparable with `memcmp`. The larger the data structure, the less likely it is that lock-freedom can be guaranteed.
- **`atomic_flag` for Boolean operations**
The only data type guaranteed to be lock-free—but only for the very few operations it supports.

29.7.1 Overview of the Operations

`atomic<T>` is not copyable or assignable in the classical sense. When you assign, do not do so with another `atomic<T>`, but with a `T`—that is, the contained value.

Here is a rough overview of the indivisible operations supported by `atomic<T>`:

- **load or operator `T`**
Reading the stored value; the latter is the implicit conversion to the enclosed type. For an `atomic<bool> a`, `if(!a.load())...` is equivalent to `if(!a)...`
- **store or operator `=`**
Stores a value. For an `atomic<bool> a`, `a.store(true)` is equivalent to `a = true`.
- **exchange**
Sets a new value and returns the old one.
- **`compare_exchange_weak` and `compare_exchange_strong`**
If `o` is the currently stored value in `atomic<T> a`, then `a.compare_exchange_strong(e, n)` roughly executes: `if(o==e) { a=n; return true; else return false; }`. In this way, you can implement something like “store a new value only if no one else has changed it in the meantime.” This way, you can prevent accidentally overwriting something. Note that the comparison is done with `memcmp` and the assignment with `memcpy`, not with potentially overloaded operators.
- **Arithmetic with `+=`, `-=`, `--`, and `++` as well as `fetch_add` and `fetch_sub`**
The operators perform the usual calculations. The `fetch_....` methods add or subtract and simultaneously return the old value.

- **Bitwise arithmetic with `|=`, `&=`, and `^=` as well as `fetch_or`, `fetch_and`, and `fetch_xor`**
Performs the usual bitwise arithmetic operations. The `fetch_...` methods return the old value.
- **Flag operations `test_and_set`, `clear()`, `test()`**
`atomic_flag` can perform exactly these two methods and no others. For this reason, they are also guaranteed to be lock-free. `clear` sets the stored value to `false`. `test_and_set` sets the stored value to `true` and returns the previously stored value. The `test()` method, which only reads the current value without simultaneously changing it, is available from C++20.

It depends on `T` whether the individual operations are lock-free. Only for `atomic_flag` with its few narrow methods is the guarantee always given.

You can find out if the operations of an `atomic<T>` are implemented lock-free by calling `is_lock_free()`. An example is shown in [Listing 29.40](#).

```
// https://godbolt.org/z/dG318o9ez
#include <iostream>
#include <atomic>

struct CArray { int a[100]; };
struct Simple { int x, y; };

int main() {
    std::atomic<CArray> carray{};
    std::cout << (carray.is_lock_free() ? "lock-free" : "locks")
              << '\n';           // Output: locks
    std::atomic<Simple> simple{};
    std::cout << (simple.is_lock_free() ? "lock-free" : "locks")
              << '\n';           // Output: lock-free
}
```

Listing 29.40 How to determine if the operations are lock-free.

For example, on MSVC 19 and GCC 14 or Clang 18 on x86-64, `atomic<Simple>` is lock-free, but the large `atomic<CArray>` is not.⁵

29.7.2 Memory Order

When you start with `atomic`, you should begin with `memory_order_seq_cst`. This is also the default for all operations. On the other hand, if you have already decided to use `atomic`, you are aiming for maximum performance. And then you should also keep an

⁵ Use additional `-latomic` flags when compiling with GCC or Clang.

eye on the other options you can choose for the constraints on the operations of `atomic`:

- **`memory_order_seq_cst`**

This is the default if you do not specify any additional parameters for the memory order in the methods of `atomic<T>`. However, it is the most restrictive rule. It states that other operations on this `atomic` must be completed before or after. `seq_cst` stands for *sequentially consistent*. So, for a given execution of operations on an `atomic a` in multiple threads, you could still relate each one to others in such a way that you can say it happens before or after. This is called a *single total order*.

- **`memory_order_acquire` and `memory_order_release`**

Typically, you mark a write operation with `release` and a read operation with `acquire`. These are then brought into a before-and-after relationship but do not restrict others.

- **`memory_order_relaxed`**

For operations marked this way, you do not set any before-and-after rules. This allows for potentially the fastest code but requires other synchronization mechanisms and is therefore definitely intended only for experts.

- **`memory_order_acq_rel`**

Some operations that read, modify this value, and then write it back can be marked this way. This is probably rare outside the implementation of the standard library.

- **`memory_order_consume`**

This operation is intended for optimizing chained data structures and when for you mostly read and rarely write from the data structure. A typical case is this: If you want to dereference a pointer `p`, you do so with `memory_order_consume`. You mark changing the pointer with `memory_order_release`. This way, readers do not interfere with each other, and writers do not break anything. Because this is a narrowly defined special case, it is only for experts.

On the x86 architecture, a memory order specified by you only really affects the `store` operation. This means, on the one hand, that detailed optimization is rarely worthwhile, and on the other hand, that you need to check the impact on errors and performance on an architecture like ARM if you use anything other than `memory_order_seq_cst`.

To give you an idea of what you could theoretically gain if you could get rid of a lock you introduced with optimal memory order specifications, I refer you to the talk “The Sad Story of Memory-Order-Consume” given by Paul E. McKenney (<https://youtu.be/ZrNQKpOypqU?t=4m>). Some operations in a CPU register take less than one clock cycle. Reading from memory can be 30 times slower. A simple additional lock for synchronization takes between twice to ten times as much time. You have achieved a lot if you can get your read/write operations distributed across two threads as fast as memory access. Faster is an absolute exception.

29.7.3 Example

Using a short example, I would like to demonstrate both the general usage of `atomic` and the memory order options.

The `SpinlockMutex` class implements a mutex that you can use together with one of the lock classes. The advantage over a normal mutex is that the thread is not put to sleep when it tries to lock the `SpinlockMutex`. Instead, it waits in a tight loop for the lock to be released. This approach is a good idea if you expect to wait only a very short time for the lock to be released. Putting a thread to sleep and waking it up again takes time. If the mutex is unlocked after just one or two loop iterations, this is better.

```
// https://godbolt.org/z/EnjPr5r76
#include <atomic>
class SpinlockMutex {
    std::atomic_flag flag_;
public:
    SpinlockMutex()
        : flag_{ATOMIC_FLAG_INIT}
    {}
    void lock() {
        // e.g., called by lock_guard
        while(flag_.test_and_set(std::memory_order_acquire)) // mostly read
            { /* nothing */ }
    }
    void unlock() {
        flag_.clear(std::memory_order_release); // write operation
    }
};
```

Listing 29.41 The “SpinlockMutex” class prevents sleeping while waiting.

If you use such a `SpinlockMutex m`, for example, with `lock_guard<SpinlockMutex> g{m}`, then it calls `lock`. The while loop then tries to set the flag until it succeeds. Since this happens in a very tight loop, the CPU is indeed stressed. With `memory_order_acquire`, you ensure that this is classified as a read operation and *after* a `memory_order_release` of another thread.

This is exactly what happens in `unlock`. There, the flag is simply deleted. This write operation is marked with `memory_order_release` and thus occurs before read operations from other threads.

The advantage of using memory order options over the default `memory_order_seq_cst` here is that other read and write operations are allowed to occur simultaneously.

29.8 Coroutines

Coroutines are a way to run multiple functions quasi-parallel within a thread. A caller interrupts its current work by starting a coroutine. The coroutine then runs until it interrupts itself—cooperatively. The coroutine can produce an intermediate result and pass it to the caller. Later, the caller can decide that the coroutine should continue its work. And so it can go back and forth.

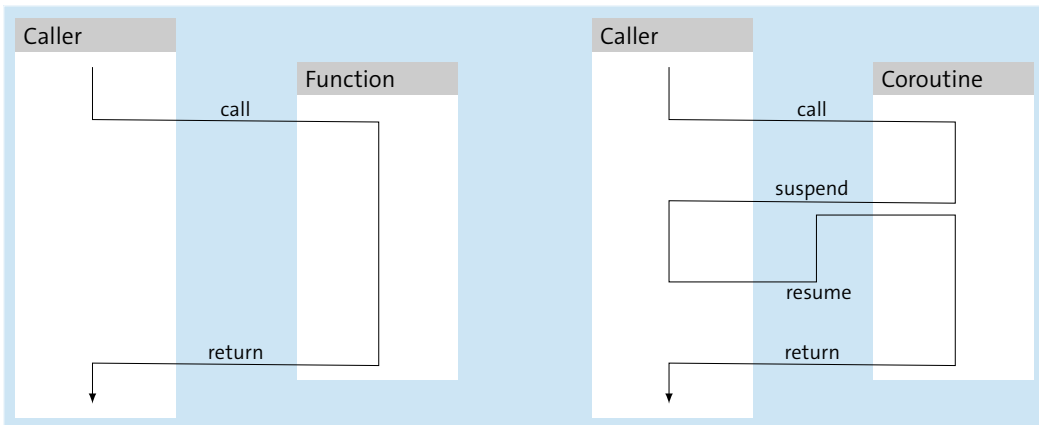


Figure 29.1 Functions only return; coroutines pause and resume.

There are various application patterns for coroutines. For example, you can implement a generator. The coroutine continuously generates values while the caller retrieves the values. But communication between different parts of the program is also possible. These are sometimes called *channels*. The same coroutine can also be controlled by multiple threads.

C++20 only provides the framework for dealing with coroutines. To actually use them, you currently have to implement the infrastructure yourself, with one exception: the standard library in C++23 provides the *generator* class as an application case for coroutines. It is planned to include more tools for coroutines in future C++ versions.

29.8.1 Coroutines in the Compiler

A *coroutine* is a function that contains the keyword `co_return` instead of `return` and can optionally also contain `co_yield` and `co_await`:

- **co_return**

This defines the final return value of the coroutine instead of `return`. The caller can then continue its execution path.

- **co_await**

This allows the coroutine to pause its work and promises that it can resume its work, for example, when an intermediate result it needs is available.

- **co_yield**

This produces an intermediate result from the coroutine and suspends it at the same time. In principle, it is a combination of `co_await` and `co_return`.

A coroutine is started with a normal call. The return value of a coroutine is a class that must have certain properties. Unlike in languages like Java, this is not a base class from which you must derive. It must simply have the correct properties (traits):

- **promise_type**

This controls the behavior of the coroutine, as it contains many configuration points (*customization points*). Typically, `promise_type` is a struct or type alias in the returned class. Depending on the use case, you implement these accordingly.

- **coroutine_handle**

When the coroutine is created, a handle of this type is generated using the `promise_type`. Typically, it is a field of the returned class. With the handle, you can control the coroutine—that is, resume and terminate it.

- **Custom interface**

The return class can contain additional data and methods needed for communication from the caller to the coroutine and vice versa. For example, this could be `resume(...)` or `operator()(...)`.

Depending on what you implement, the coroutine behaves differently. For example, it can deliver a value or not, receive values or not; it can start immediately or wait until it is explicitly started; it can be started multiple times or only once. You can also influence the behavior in case of an exception.

You can write your own classes that you equip with a corresponding `promise_type`. In doing so, you can use APIs that offer their functionalities via coroutines. Then you only need to operate them from the outside or implement the coroutines yourself.

29.8.2 Generator

The simplest case is `std::generator<>`, which is included starting from C++23. Simply write a coroutine that returns this template class, and the compiler does the rest. The `generator` class is conveniently a range adapter (view), so you can use it, for example, with `for` loops or combine it with `operator|`.

```
// https://godbolt.org/z/Gfazq6GE7
#include <generator>
#include <iostream>
#include <vector>

std::generator<int> fib(int n) { // generates int values
    int a = 0, b = 1;
    while (--n) {
```

```
    co_yield b;                                // makes this function a coroutine
    auto tmp = a;
    a = b;
    b += tmp;
}
}
int main() {
    for (auto i : fib(10)) std::cout << i << ' ';
    std::cout << '\n';                        // Output: 1 1 2 3 5 8 13 21 34 55
}
```

Listing 29.42 Your coroutine returns a “generator”.

Because `fib` contains a `co_yield`, the function becomes a coroutine. The return type is `std::generator<int>`, and all the magic is contained within. There you will find the `promise_type`, a `coroutine_handle`, an iterator, and much more.

With `fib(10)`, you enter the coroutine. It starts immediately because the `promise_type` of `generator` is configured that way. As soon as the compiler reaches `co_yield`, the coroutine is paused and the value `b` is stored in `generator`. Control is then returned to the caller. The `for` loop retrieves the `int` from the generator and outputs it.

The `for` loop now increments the iterator, causing the coroutine to resume. The same process occurs at the next `co_yield` until—in this case—10 values have been delivered. The coroutine terminates and is cleaned up (because `promise_type` dictates so).

29.8.3 Coroutines with “`promise_type`”

Custom return types with `promise_type` are somewhat more complicated. You need to write the class yourself and implement the properties you need.

Your return type must be a class that contains or names a `promise_type`. This `promise_type` must have the following methods:

- **`get_return_object()`**
This method is called when the coroutine is started. It returns an appropriate `std::coroutine_handle`. With this, the caller communicates with the coroutine.
- **`initial_suspend()`**
Returns either `suspend_always` if the coroutine should not start immediately or `suspend_never`.
- **`final_suspend()`**
Contains final cleanup operations of the coroutine. Returns either `suspend_always` or `suspend_never`.

■ `unhandled_exception()`

Called when an exception occurs in the coroutine. It can handle this exception or rethrow it.

One of the simplest possible implementations looks like the following listing:

```
// https://godbolt.org/z/8enx43Ksb
#include <coroutine>
#include <iostream>

struct Routine { // Coroutine return type
    struct promise_type; // defined further ahead
    Routine(auto h) : hdl_(h) {} // Constructor
    ~Routine() { if(hdl_) hdl_.destroy(); } // Destructor
    Routine(const Routine&) = delete; // no copies
    Routine(Routine&&) = delete; // no moves
    // Interface for the caller:
    bool more() { // continue coroutine
        if(!hdl_ || hdl_.done()) return false; // - nothing more to do
        hdl_.resume(); // - blocking call
        return !hdl_.done(); // - signals done or not
    }
private:
    using handle_type_ = std::coroutine_handle<promise_type>;
    handle_type_ hdl_; // to control the coroutine
};

struct Routine::promise_type { // necessary promise_type
    auto get_return_object() { // initializes the handle
        return Routine{ handle_type_::from_promise(*this) };
    }
    auto initial_suspend() { return std::suspend_never{}; } // start immediately
    auto final_suspend() noexcept { return std::suspend_always{}; } // normal case
    void unhandled_exception() {} // no exception handling
    void return_void() {} // Coroutine has no return
};

Routine counter() { // Return type is our wrapper class with the promise_type
    for (unsigned i = 0;; ++i) {
        co_await std::suspend_always{}; // Pause coroutine
        std::cout << "counter: " << i << std::endl;
    }
}

int main() {
    Routine routine = counter(); // Create and start coroutine
    for (int i = 0; i < 3; ++i) {
        std::cout << "Hello main!\n";
    }
}
```



```
        routine.more();                // continue coroutine
    }
}
```

Listing 29.43 A simple example of a coroutine return type with its “`promise_type`”.

The `Routine` class is the wrapper for the `coroutine_handle`. It also contains the `promise_type`, which describes the behavior of the coroutine. In `get_return_object()`, the handle is initialized and returned to the caller. The rest of the methods are trivial.

The caller receives the return value from `get_return_object()` when starting. This is essentially the factory method for the return type. The constructor of `Routine` receives the handle and stores it for later communication with the coroutine. The destructor calls `destroy()` on the handle if it is still present. Here, this is always the case, but with move semantics, it can become important because then the handle can also be empty.

The coroutine is resumed with a call to `more()` by invoking `resume()` on the handle. `hdl_()` would have achieved the same effect, by the way. Beforehand, we check with `!hdl_` whether a handle is present at all. And we check with `hdl_.done()` whether the coroutine is already finished. In our specific coroutine, this is never the case, but in general, it can happen. The return value is simply passed through from `hdl_.done()` and signals to the caller whether the coroutine could continue running.

With the return value of `initial_suspend()`, you determine whether the coroutine should start running immediately upon creation or not. With `suspend_never`, it does not start yet (*lazy*); with `suspend_always`, it would start immediately (*eager*).

What `final_suspend()` returns determines what happens at the end of the coroutine's lifetime. Here, `suspend_always` is the normal case.

The really interesting definition is the coroutine `counter()`. Its return type is `Routine`, which is the wrapper. It pauses three times with a call to `co_await suspend_always{}`. With that, program control returns to the caller. The caller then continues the loop and calls `more()`. This causes the coroutine to resume until the next `co_await`.

Awaiter and Awaitable

The `suspend_always` and `suspend_never` types are so-called awaitables. They are the simplest possible awaitables and are provided by the library. These are classes where the compiler expects you to implement the methods `await_ready()`, `await_suspend()`, and `await_resume()`. Normally, you don't have to do this. With *awaiters*, you also get values back into the coroutine.

Because we don't return any values with `co_yield` or `co_return`, we define the method `return_void()` in the `promise_type`. If we were dealing with values, a different method would be here. For example, if we used `co_yield 54`, it would be `auto yield_value(int)`, and for `co_return 73L`, it would be `auto return_value(long)`.

The interface of `promise_type` is intentionally kept general to suit many requirements. This does not make the first steps with coroutines easy, but there will surely soon be libraries built on `promise_type` that can offer you a simple interface for various requirements.

29.9 Summary

Here I summarize the elements for multithreading in the standard library in a brief overview:

- Avoid having to manage threads yourself wherever possible.
- Use `jthread` for threads (from C++20; otherwise, `thread`).
- Use `stop_token` as a cancellation mechanism (from C++20).
- Use `latch` and `barrier` (from C++20) and possibly `future` for synchronization.
- For the rest, use `mutex`.
- Use `atomic` only in the rarest cases.

29.9.1 “<thread>” Header

- `thread`, `jthread`—`join()`, `detach()`, `joinable()`, `get_id()`, `native_handle()`
Classes for starting threads. `join()` waits for the thread to finish. `detach()` allows the thread to continue running even if the `thread` object is destroyed. `get_id()` returns a unique thread identifier. `native_handle()` allows platform-dependent interaction with a thread.
- `this_thread::`—`get_id()`, `yield()`, `sleep_for`, `sleep_until`
Namespace with some helper functions for information and manipulation of the current thread. `get_id` is used to identify the current thread. `yield` lets other threads take their turn. `sleep_for` and `sleep_until` pause the thread for a specified time.
- `thread::id`—`operator<<`, `operator<` etc., `hash<thread::id>`
Type for uniquely identifying a thread. Can be used in associative containers and output to a stream thanks to overloaded operators.

29.9.2 “<latch>” and “<barrier>” Headers

- `latch`—`count_down()`, `wait()`, `try_wait()`
One-time use latch for threads. `count_down()` decreases the counter. `wait()` blocks until the counter is 0 (since C++20).
- `barrier`—`arrive_and_wait()`, `arrive_and_drop()`, etc.
Reusable barrier for thread synchronization. `arrive_and_wait()` blocks until all threads have arrived at the barrier (since C++20).

29.9.3 “<mutex>” and “<shared_mutex>” Headers

- **mutex—lock, unlock, and try_lock**
Base class for protecting a critical section. `lock()` blocks before a critical section until the thread gets its turn. `unlock()` gives other threads access to the critical section. `try_lock()` does not block, but returns `true` on success.
- **timed_mutex—also try_lock_for and try_lock_until**
Protection with a time limit for blocking. The methods block only for a certain time.
- **recursive_mutex**
Allows `lock()` to be called multiple times in succession by the same thread. The protection is only released when `unlock()` has been called the same number of times.
- **recursive_timed_mutex**
Combines the properties of `timed_mutex` and `recursive_mutex`.
- **shared_timed_mutex—in <shared_mutex>, also lock_shared() and others**
Allows, for example, a writer to exclusively lock while multiple readers can share a lock.
- **lock_guard**
Allows locking and unlocking of a mutex in RAII style.
- **scoped_lock**
Locks multiple mutexes simultaneously in RAII style, thus preventing deadlocks.
- **unique_lock**
A mutex that can be moved similarly to a `unique_ptr`.
- **shared_lock**
Allows both exclusive and nonexclusive locking of a `shared_timed_mutex`.
- **lock and unlock—free functions**
Locks and unlocks one or more mutexes in a non-RAII manner.
- **once_flag and call_once**
Allows a function to be executed at most once.

29.9.4 “<condition_variable>” Header

- **condition_variable—notify_one, notify_all, wait, wait_for, and wait_until**
For communication between multiple threads using a `unique_lock`, in RAII style. With `notify`, waiting threads are informed about a possible change of a `condition_variable`. The `wait` methods wait for such a change.
- **condition_variable_any**
Communication between threads with any kind of lock.
- **notify_all_at_thread_exit**
Helper function to inform other threads about the end of a thread of a `condition_variable`.

29.9.5 “<future>” Header

- **async—free function**
Starting a new thread or deferring the execution of a computation; returns a future.
- **future and shared_future—get(), valid(), wait..., and share()**
Encapsulates the result of a computation that is retrieved only upon get(). valid() checks if the result is already available. wait, wait_for, and wait_until wait until the result is available or for a specified duration. shared() converts a future into a shared_future.
- **promise—get_future, set_value, set_exception, ...at_thread_exit**
Manually creates a future when async is not used.
- **packaged_task—operator(), get_future(), valid(), reset**
Prepares a function for later execution and provides the result via future.

29.9.6 “<atomic>” Header

- **atomic<T>, atomic_ref<>, atomic<shared_ptr<>>, atomic_flag, atomic_int, etc.—load, store, exchange, etc.**
Data types whose methods cannot be interrupted by other threads. For all built-in data types and many of their aliases, there is a predefined atomic equivalent. atomic_ref and the specialization for shared_ptr and floating-point types are available from C++20.
- **memory_order—from cst_seq to relaxed**
Depending on the use case, optional parameters for the methods of the atomic data types.
- **atomic_thread_fence, atomic_signal_fence**
Creates a barrier in the machine code that you can use to synchronize non-atomic data structures similarly to an atomic or to synchronize with one.
- **atomic_load, atomic_store, atomic_exchange, etc.—free functions**
Interaction of a nonatomic data type with an atomic one.

29.9.7 “<coroutine>” Header

For writing your own coroutines and collaborating with co_yield, co_await, and co_return (from C++20):

- **coroutine_handle**
Pointer to a coroutine that was interrupted with co_await or co_yield.
- **coroutine_traits**
The way to get from a return type to the appropriate promise_type.
- **suspend_never, suspend_always**
Helper classes for collaborating with co_await and co_yield.

Chapter 30

Good Code, 7th Dan: Guidelines

Chapter Telegram

■ C++ Core Guidelines

Collection of guidelines for programming good C++ code; maintained by C++ users, initiated and managed by Bjarne Stroustrup and Herb Sutter.

■ Guideline Support Library (GSL)

Part of the C++ Core Guidelines. Defines a few types whose use primarily allows automated tools to detect typical errors. There is at least one implementation of the GSL and at least one tool that already uses it.

C++ is a comprehensive language. It intentionally gives programmers a lot of freedom so they can get the most out of the language if necessary. It allows pursuing your own approaches and exploring new paths more than other languages.

The downside to this freedom is that not everything always happens automatically. For many features, there is a recommended way that is most likely to lead to success. Based on many programming experiences already gathered, the best way has crystallized for one thing or another. It's worth knowing the recommendations of the experts to benefit from this knowledge.

In this book, I tried to give you an idea of what I call *modern C++*. I talked about the advantages of `RAII` and `const`, as well as how to use the standard library. There's more to it than that, though. You can find lots of other tips in the literature. I want to mention one new development in this chapter. Bjarne Stroustrup, the creator of C++, and Herb Sutter, a top Microsoft C++ expert, have launched an initiative to develop the *C++ Core Guidelines*. I think this is a great idea and have been following it closely.

There are other collections of tips, other guidelines, and many may be correct, deeper, better, or worse. The special thing about the C++ Core Guidelines is that they have the potential to become a quasi-standard and can serve as a foundation for further guidelines.

In this chapter, I provide an overview of the guidelines. Otherwise, I recommend reading the complete C++ Core Guidelines, or at least a selection of them. There you will also find explanations, examples, and discussions. Unfortunately, full explanations do not fit in this book; they would take up their own 500 pages. At the end, I give specific tips on some topics and list the most relevant rules.

30.1 Guideline Support Library

The “GSL: Guideline Support Library” section of the C++ Core Guidelines introduces some new language elements that help in adhering to the rules. These are things that

- are particularly common sources of errors, and
- would be particularly difficult to detect with automated tools.

Microsoft already provides an implementation of this GSL.¹ It is a collection of headers that you can simply include in your program with `#include <gsl/gsl>`. If you do this, Visual Studio also offers a plug-in that can help you spot a few typical errors.

Here you will find an excerpt of the GSL constructs. These are consistently designed so that they do not negatively impact runtime performance. In some cases, such as with `owner`, they are merely a kind of semantic annotation and actually do “nothing”:

- **`owner<T*>`**
Use this owning raw pointer in appropriate places instead of `T*`.
- **`not_null`**
This marks raw pointers that can never be `nullptr`.
- **`Expects()` and `Ensures()`**
Used in functions for the formal definition of preconditions and postconditions.
- **`finally(...)`**
This helper function creates an object that is executed at the end of the current code block and is more suitable than an extra `try-catch` block if you do not want to write your own RAII wrapper.

The GSL is still in flux and is based on the C++ Core Guidelines. You can find motivation and introduction by Herb Sutter.²

In short, when using the bare C++ language, it is impossible to prove the correctness of a given program regarding type safety and resource correctness or to have it calculated by an automatic tool. The GSL is intended to make this possible. With the constructs from the GSL, you enable a tool to do this automatically. The GSL itself is platform-independent. Microsoft has integrated it into Visual Studio, and there is a plug-in that helps in checking the C++ Core Guidelines along with the GSL.

30.2 C++ Core Guidelines

To apply the guidelines, it is important to understand the intent of the guidelines’ authors. In this section, I will describe the “how and why” of the guidelines. After that, I

¹ <https://github.com/Microsoft/GSL>, [2023-12-03]

² *Writing Good C++14... By Default*, Herb Sutter, <https://www.youtube.com/watch?v=hEx5DNLWGgA>, 2015, [2017-05-14]

will take a problem-oriented look at some aspects of everyday programming and apply specific guidelines to them.³

Each guideline has an identification. This consists of one (or more) letters and a number—for example, *F.22*. The letter determines the section from which the guideline originates. The section *F* refers, for example, to *functions*.

The sections are further divided into subsections. For example, guidelines F.1 to F.9 together form the *F.def* subsection.

Ahead, I will use the identifications of the guidelines and subsections so that you can easily find them again. I will quote some selected guidelines. I cannot list the entire explanation of any guideline here, but I have added a few explanatory words [in brackets] at some points.

30.2.1 Motivation

I can hardly say it better than the editors of the guidelines, so here I provide you with an excerpt of the abstract.

Abstract⁴

This document is a set of guidelines for using C++ well. The aim of this document is to help people to use modern C++ effectively. By “modern C++” we mean effective use of the ISO C++ standard (currently C++20, but almost all of our recommendations also apply to C++17, C++14 and C++11). In other words, what would you like your code to look like in 5 years’ time, given that you can start now? In 10 years’ time?

The guidelines are focused on relatively high-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast—you can afford to do things right.

We are less concerned with low-level issues, such as naming conventions and indentation style. However, no topic that can help a programmer is out of bounds.

Many of the rules are designed to be supported by an analysis tool. Violations of rules will be flagged with references to the relevant rule. We do not expect you to memorize all the rules before trying to write code. One way of thinking about these guidelines is as a specification for tools that happens to be readable by humans.

The general goal of the guidelines is to create a codebase over time that is shared by multiple developers worldwide. A style should emerge that looks similar here and

³ <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>, [2023-12-03]

⁴ <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>, [2024-06-02]

there and pursues similar goals. This way, everyone can more easily familiarize themselves with new projects over time, and their code is also quicker to read for others, and thus self-documenting. Adhering to the guidelines partially fulfills itself, as one guideline is that code should document itself as much as possible.

However, the guidelines should not force anyone to use only a part of the language, nor is code that does not follow the guidelines necessarily bad. The primary guideline is still that C++ is expressive and performant, and an author must be able to exploit this when it is right and important.

But if you follow the guidelines, you can expect your code to become more correct, safer, and more maintainable without becoming slower.

30.2.2 Type Safety

Many of the guidelines aim to make your life easier overall by relying more on the static type system of C++.

Do Not Weaken the Type Safety of C++

Type safety is weakened by the use of `void*` or `reinterpret_cast`, with limitations by other forms of explicit type conversions, and by (unmarked) `union`.

If you need to use one of these-mentioned operations because a third party requires it, then hide it behind a layer maintained by you. Do not let `void*` spread in your code.

Avoid “Type Dumpsters”

Do not use `string` and `int` for everything. Be more specific by introducing, for example, `Month`.

Trick question: What does the following function do?

```
Color createColor(int a, int b, int c);  
Color createColor(int a, int b, int c, int d);
```

There is no correct answer here. You should make the interface cleaner, for which there are many possibilities:

```
Color createColor(Red r, Green g, Blue b);  
Color createColor(Cyan c, Magenta m, Yellow y, Black k);
```

This is even better than just naming the parameters `r`, `g`, and `b`. This way, you can hide in the data type whether the red component is specified from 0 to 100, from 0 to 255, or even as a double between 0.0 and 1.0.

You avoid accidentally swapping the components this way. And what if you actually wanted to call `createColor(a,b,c,d)` but forgot a component and instead called `createColor(a,c,d)`? Then you have called the completely wrong function.

Relevant Guidelines

The relevant guidelines here include the following:

- P.1: Express ideas directly in code.
- P.4: Ideally, a program should be statically type safe.
- I.1: Make interfaces explicit.
- I.24: Avoid adjacent arguments of the same type (...).
- Type.1: Avoid casts. (Especially do not use `reinterpret_cast`.)
- Type.2: Do not use `static_cast` to downcast. Use `dynamic_cast` Instead.
- Type.3: Do not use `const_cast` to cast away `const` (i.e., at all).
- Type.4: Do not use C-style `(T)expression` or functional `T(expression)` casts: prefer construction or named casts or `T{expression}`.

Use `std::chrono` and `std::time_point` for time units. You can also use the type-safe `boost::units` for calculations with numerical units.

30.2.3 Use RAII

Write constructors in such a way that

- when you create an object, you can always clean it up in the destructor; and
- if you throw an exception, no resources remain allocated.

So it must always be possible to create either a complete object or none at all. A complete object can be one that holds an error state. The main thing is that it is correctly cleaned up in the destructor.

I have advocated for this multiple times in this book. Construction and destruction of objects are explained in detail in [Chapter 16](#). The problems with raw pointers and their alternatives are discussed in [Chapter 20](#).

All or None: The Default Operations

As a reminder, C++ defines the following *default operations*:

- A default constructor: `X()`
- A copy constructor: `X(const X&)`
- A copy assignment: `operator=(const X&)`
- A move constructor: `X(X&&)`
- A move assignment: `operator=(X&&)`
- A destructor: `~X()`

You make it easiest for yourself if you follow the *rule of zero*, meaning neither defining nor deleting the default operations with `=delete`. Let the compiler do its job and generate the default operations itself.

Of course, this only works if the compiler-generated default operations do what they are supposed to do. That means you should correctly initialize, copy, and move, and not leave any resources behind.

This, in turn, is easiest if you only use data fields that follow those rules themselves. Built-in types, aggregates, standard containers, smart pointers, and most other data structures of the standard library adhere to that. Raw pointers `T*` and references `T&`, which own their objects, do not.

Of course, some classes—those that connect to an external resource—require special attention, starting with the destructor. If that is the case, then the rule is this: define or delete *all* default operations (if possible or reasonable, also the default constructor). In practice, you need a custom-written destructor exactly when you have to manage a resource yourself. And if that is the case, this resource must also be accounted for when copying, moving, and assigning.

Do You Know the Guidelines for Construction, Assignment, and Moving?

However, if you need to define the default operations yourself, keep in mind that implementing them correctly is possible but not trivial. You should rely on the compiler whenever possible.

If you still need to define the operations, don't forget `noexcept`, don't let any exception leave a destructor, and use `std::owner` to mark owning raw pointers.

Pitfall: Noncreated Object Occupies Resources

If the constructor exits with an exception, an object is considered not created. The destructor is then not called. Ensure that despite the exception, all resources allocated beforehand are released again.

Pitfall: Created Object Despite Exception

You avoid code duplication by using delegating constructors. But remember that an object is considered created as soon as the “innermost” constructor has been run. If a *delegating* constructor throws an exception afterward, the *destructor will still be called!*

```
// https://godbolt.org/z/veEq4dcET
struct Day {
    Day(int a, int b) : Day{} {           // delegates
        if(a==0 || b == 0)
            throw 666;                   // throws exception
    }
    Day() {}
```

```
};
int main() {
    try {
        Day day{1,2};
    } catch(int) { }
}
```

Here, the first constructor delegates part of the initialization tasks to the constructor `Day()`. That's fine because whatever initialization needs to be done can be centrally maintained this way. You just need to be aware that with the `throw 666` exception, the `Day()` constructor has already been completely run and the object is thus considered constructed. This means the destructor will be called. This is not a problem in this simple example, but it can become one in more complicated cases.

Keep in mind that in the presence of delegating constructors, not every exception means that the destructor will not be called.

Relevant Guidelines

These run through all guidelines and sections. Here are some of the most obvious and relevant ones (with my annotations in square brackets):

- P.8: Don't leak any resources.
- F.22: Use `T*` or `owner<T*>` to designate a single object. [That means `T*` should not be used for pointer arithmetic. This is good for automated tools too.]
- Sections C.defop, C.dtor, C.ctor, and C.copy. [Here you will find plenty of tips on (whether and) how to implement the default operations.]
- R.1: Manage resources automatically using resource handles and RAII.
- R.3: A raw pointer `T*` is nonowning. [`T* p` does not own the object, so `delete p` should be avoided; instead, `owner<T*> p` with `delete p` is fine, etc.]
- R.4: A raw reference `T&` is nonowning. [No `int& r = *new int{7}` followed by `delete &r`.]
- R.5: Prefer scoped objects; don't heap-allocate unnecessarily. [Not with `new` or `malloc` or otherwise.]
- R.20: Use `unique_ptr` or `shared_ptr` to represent ownership.
- R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership.
- E.6: Use RAII to prevent leaks.
- E.13: Never throw while being the direct owner of an object [and thus bypass the intended `delete` (or other cleanup operations)].
- E.16: Destructors, deallocation, and `swap (...)` must never fail. [So, neither throw an exception nor fail.]
- Section C.other mainly deals with value classes. [It explains what you should consider for `swap` and `==`.]

30.2.4 Class Hierarchies

Casting a UML diagram into an implementation is not a one-to-one transformation, even if it is a class diagram. There are implementation details to consider. Conversely, you can represent a given C++ class hierarchy with a UML diagram. Not every method needs to be implemented as a member. An (overloaded?) free function is better suited for extension if you do not need to access the internals of the class.

No, a good class hierarchy is not an easy task. And because software development is a moving target, you must ensure during the design phase that your hierarchy is testable so that you can respond to changes in requirements with new extensions and refactorings of existing components. There are also helpful guidelines for this.

You will find a lot about this in this book—concisely in [Chapter 19](#), but also in the more technical chapters like [Chapter 15](#).

Multiple Inheritance Is Difficult but Powerful

With multiple inheritance, you can map design patterns recognizably and save a lot of code duplication and work. You can make your components more loosely coupled, modular, and extensible.

Multiple inheritance is also a curse in several respects: misused, your code becomes opaque and rigid. Avoid having code from all possible directions converge in a single class. In practice, a distinction is often made between classes that contribute to the implementation and those that provide the interface.

You should have a main hierarchy that focuses on the implementation. This way, you avoid code duplication and use the technical tool of inheritance to improve your code.

Otherwise, you should use *interface classes* to declare the interfaces. You can then provide the methods of these interfaces—for example, via *delegation*. This means you write a class that implements the interface, from which you then embed an instance in the actual class as a member and forward all calls to it. Sometimes these delegated classes are very specialized and are only needed once. Nevertheless, you separate tasks and responsibilities. Your code becomes more object oriented.

Virtual or Not?

Technically speaking, calling a virtual method is only slightly more expensive than calling a nonvirtual method. However, there are other disadvantages that should prevent you from making all methods virtual by default:

- At first glance, calling a virtual method is only associated with an additional indirection through the instance's *virtual method table*.
- A virtual method can extremely rarely, if ever, be made into an inline method by the compiler.

- Besides the non-`inline` code of the method, the function call hinders any cross-function optimization by the compiler *and* acts as a time-consuming synchronizing barrier in multithreading.
- The pointer to the virtual method table is not negligible in small (and numerous) data structures.
- A virtual method should never be called in a constructor (and should not be called in a destructor).

There is also a less technical matter to consider. What if you never override a virtual method? Well, then you shouldn't have made it virtual in the first place, right? Oh, you wanted to provide a configuration point in a library? That's good, it allows for extensibility. But now someone comes along who uses your library and actually extends it. Can the virtual method still fulfill its task? No idea, right? It's no longer your virtual method being called, but the method of the extender. This is fundamentally good, but it is a responsibility not to be underestimated. Whose? The overrider's? No—yours! You should and may only allow a method to be overridden using `virtual` if you make it difficult for extenders to misuse the class—despite the “invasive” code of an extender.

With `final`, it's even worse. When you write `final` on a method, you are implicitly saying that the behavior of the object will not change anymore. But do you also adhere to it? Do you call another virtual method in the `final` method? How can you then say that the method still does what it is supposed to do? Have you specified preconditions, postconditions, or invariants for this `final` method? You will find it difficult to guarantee these if you call other virtual methods from your `final` method.

Don't get me wrong: sometimes `final` is just a form of documentation, but technically speaking, you are on thin ice with the promise that “this method does exactly what I say” if you call another virtual method. Therefore use `final`, yes, but within limits.

Relevant Guidelines

This is a list of the guidelines most relevant to virtual methods:

- C.2: Use `class` when the class has an invariant; use `struct` when the data members can vary independently.
- C.3: Represent the distinction between an interface and an implementation by using a class. [For example, it is easier to encapsulate data and highlight methods as interfaces with a class than if you only use free functions.]
- C.8: Use `class` rather than `struct` if any member is not `public`.
- C.9: Minimize the exposure of members.
- C.35: A base class destructor should be either `public` and `virtual` or `protected` and `nonvirtual`.
- C.120: Use class hierarchies to represent concepts with an inherent hierarchical structure (only).

- C.121: If a base class is used as an interface, make it a pure abstract class [i.e., a signature class that only has pure virtual (abstract) methods; all methods `virtual` and `= 0`].
- C.122: Use abstract classes as interfaces when a complete separation of interface and implementation is needed.
- C.128: Virtual functions should specify exactly one `virtual`, `override` or `final`. [`override` and `final` imply `virtual`, so a double mention is unnecessary. On the other hand, a missing `virtual` or `override` on an overriding method is confusing (once `virtual`, always `virtual`).]
- C.130: For making deep copies of polymorphic classes, prefer a virtual clone function instead of public copy construction/assignment. [Copying a base class (with at least one virtual function) usually slices and is therefore not what you want.]
- C.132: Don't make a function virtual without reason.
- C.133: Avoid protected data. [That is, choose `public` or `private`.]
- C.135: Use multiple inheritance to represent multiple distinct interfaces.
- C.136: Use multiple inheritance to represent the union of implementation attributes.
- C.137: Use virtual base classes to avoid overly general base classes.
- C.139: Use `final` on classes sparingly.
- C.145: Access polymorphic objects through pointers and references. [This reduces the risk of accidental slicing.]
- C.165: Use `using` as a customization point. [If you use `using xyz` or `using namespace abc` in an implementation, you allow users to place their own function there.]

30.2.5 Generic Programming

When you use the standard library, you use templates because they are a powerful tool. You can potentially improve your code if you also write templates.

The easiest way is to write a *generic lambda*, which doesn't even involve the template syntax:

```
vector</*...something...*/> data;  
// ...  
... = std::ranges::find_if(data, [](auto e) { return e<10; });
```

Here `auto e` is actually a template parameter, which you would write as a function template like this:

```
template<typename T>  
auto func(T e) { return e<10; }
```

In both notations, you don't have to worry about the type of what is in the container, and you don't have to adjust it if the type changes.

The next step, to actually write your own template function if appropriate, is then not far off.

The reasons for creating your own template class are somewhat different than for a function, but here too you make your code more general and avoid code duplication. Templates also are very fast; the compiler often chooses to inline them.

And don't forget that machine code only actually ends up in the compiled output if the template function or the element of the template class is actually used somewhere in the program. This is different from nontemplate functions, which can indeed end up in the compiled program (but do not have to), even if they are not used.

Use Templates for Containers

If you write a function that sorts a specific `vector` in a specific way, you should not pass the `vector` as a concrete type. Either make it a template parameter yourself or use iterators, which is best done via template parameters.

Do not do it like this:

```
void sortDataCont(vector<int> &data) {    // ✗ entire container
    // ... exemplary ...
    sort(data.begin(), data.end());
}
void sortDataIt(vector<int>::iterator b,
               vector<int>::iterator e) {    // ✗ unnecessarily specific
    // ... exemplary ...
    sort(b, e);
}
```

Instead, define the functions as templates:

```
void sortDataCont(auto &data) {
    // ... exemplary ...
    sort(data.begin(), data.end());
}
template<typename It>
void sortDataIt(It b, It e) {
    // ... exemplary ...
    sort(b, e);
}
```

Here I used `data` to pass the entire container as a parameter in the abbreviated function template using `auto`. Here the function performs a `sort` that works on various containers.

Don't forget that you (still) need to carefully consider whether the templated parameters of the functions need to be references `&`, pointers `*`, or `consts`. Iterators are usually passed by value.

However, you can be even more flexible if you define your function for ranges:

```
// https://godbolt.org/z/Y8KK3nEG3
void sortData(std::ranges::random_access_range auto &&data) {
    std::ranges::sort(data);
}
```

Here you use an abbreviated function template via `auto` together with the concept `random_access_range`. Note that when using ranges as parameters, you should typically use the universal reference `&&`.

Universal References

When you write move operations yourself, they will receive parameters as rvalue references `&&`. When you mix this `&&` syntax with templates, you need to be aware of one thing: the `&&` parameter is not necessarily an rvalue reference, as I explained in [Chapter 23](#).

Normally, you want to forward the `&&` parameter in a template only via `std::forward` to another function (and nothing more). Or if you wanted something entirely different, then `&&` for the template parameter was probably wrong and does not do what you intended.

Relevant Guidelines

- T.1: Use templates to raise the level of abstraction of code.
- T.2: Use templates to express algorithms that apply to many argument types.
- T.3: Use templates to express containers and ranges.
- T.40: Use function objects to pass operations to algorithms. [A function object can carry more information than a simple function pointer and is usually faster.]
- T.43: Prefer using `over` typedef for defining aliases.
- T.68: Use `{}` instead of `()` within templates to avoid ambiguities. [`T v{}` always defines a variable `v`. `T v{}` could, however, be a function definition. `T{v}` is a constructor call, whereas `T(v)` could also be a cast.]
- T.69: Inside a template, don't make an unqualified nonmember function call unless you intend it to be a customization point.
- T.83: Do not declare a member function template `virtual`.
- T.144: Do not specialize function templates. [In most cases, overloading is better.]
- F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter.

30.2.6 Do Not Be Confused by Anachronisms

There are a few “nonguidelines” or rumors that have persisted for a long time and either were never true or have not been true for a long time. These have a dedicated appendix in the core guidelines. I address them here and explain them to you so that you do not fall into traps and believe rumors that you may have picked up somewhere. I needed to reword their headings a bit to emphasize that you should *not* do it:

- **NR.1. Not: All declarations should be at the beginning of a function.**

In original C and Pascal, it was required that variables be listed at the beginning of a function. It is astonishing that this impracticality can still be found in some Java or JavaScript code. No, a variable should be declared and initialized just before its use. In rare, exceptional cases, you might pull a variable out of a tight loop.

- **NR.2. Not: There should be only one “return” statement from a function.**

Yes, it used to be that certain compilers performed better when there was only a single `return` in a function. This is no longer the case. Following this rule often makes a function more complicated and the code harder to read. Error checking within the function unnecessarily bloats the function body.

- **NR.3. Not: Do not use exceptions.**

In the early days of C++, exception handling was buggy in many important implementations (e.g., in GCC versions below 2.95). Bad experiences from programmers of that time still have an effect. In addition, many believe that error handling with exceptions is slow. It is true that the presence of exceptions is the only C++ feature that violates the *zero-overhead principle*—that is, *what you do not use costs nothing*. If you generally enable exception handling (the default for all compilers) but do not throw any exceptions, your code will be minimally slower or larger (in the range of 0.5% to 2%). But then you miss out on proper RAII and a large part of the benefits of modern C++. You program around it manually, and your code becomes ... larger and slower. However, there is indeed a reason to avoid exceptions: in real-time systems that must guarantee a response time, exceptions must not occur at certain points. A *thrown* exception can potentially mean a lot of overhead, and how much that is cannot be predicted in the general case. At least in this program path, exceptions must be avoided.

- **NR.4. Not: Every class must be in its own file.**

While this is a good starting point, you should not let it prevent you from making your code even more meaningful. Also, combining several tiny classes in one file is better than having hundreds of tiny files. You can, for example, group such classes in their own namespace and have a better reason for a separate file. In C++, not everything is within classes; we have free functions that also belong in a meaningful place.

- **NR.5. Not: The constructor should do as little as possible.**

There are reasons for separating initialization in the constructor and another initialization routine. For example, the two phases might have different semantics, as is

the case in GUI programming: initialization of data and initialization of views. However, the reason should *not* be that there is too much code or it takes too long. The most important thing is that you throw an exception when an error occurs so that the object is considered not constructed. A small or large constructor is suitable for this. It is also not the case that object creation is done by a central thread and other threads are blocked.

■ **NR.6. Not: Put all cleanup actions at the end of a function and use goto exit.**

This technique applies to C, but no longer to C++ in the presence of RAII. If you don't use RAII, use `gsl::finally`.

■ **NR.7. Not: Make all data fields protected.**

Either a field is public or it is private. If you make something `protected`, then you have no control over who manipulates this data field, bypassing all access mechanisms, because someone can derive from your class. And thus the supposedly `protected` data field is as good as `public`. The nonguideline that a data field should definitely be private and only accessible via a getter and setter also applies here. Don't do that in C++. Don't write a trivial getter and setter just to make a public field private. Instead, simply make the field public.

Appendix A

Cheat Sheet

The following pages contain a cheat sheet that you can use to look up containers and their operations.

Operation	array	vector	deque	forward_list	list	set
cbegin/cend	contig.	contig.	rand. access	→	↔	↔
crbegin/crend	contig.	contig.	rand. access		↔	↔
assign		assign	assign	assign	assign	
at	at	at	at			
operator[]	operator[]	operator[]	operator[]			
front	front	front	front	front	front	
back	back	back	back		back	
empty	empty	empty	empty	empty	empty	empty
size	size	size	size		size	size
max_size	max_size	max_size	max_size	max_size	max_size	max_size
resize		resize	resize	resize	resize	
capacity		capacity				
reserve		reserve				
shrink_to_fit		shrink_to_fit	shrink_to_fit			
clear		clear	clear	clear	clear	clear
insert		insert	insert	insert_after	insert	insert
emplace		emplace	emplace	emplace_after	emplace	emplace
emplace_hint						emplace_hint
erase		erase	erase	erase_after	erase	erase
push_front			push_front	push_front	push_front	
emplace_front			emplace_front	emplace_front	emplace_front	
pop_front			pop_front	pop_front	pop_front	
push_back		push_back	push_back		push_back	
emplace_back		emplace_back	emplace_back		emplace_back	
pop_back		pop_back	pop_back		pop_back	
swap	swap	swap	swap	swap	swap	swap
merge	<algo>	<algo>	<algo>	merge	merge	<algo> ³
splice				splice_after ¹	splice	
remove	<algo>	<algo>	<algo>	remove	remove	
remove_if	<algo>	<algo>	<algo>	remove_if	remove_if	
reverse	<algo>	<algo>	<algo>	reverse	reverse	
unique	<algo> ²	<algo> ²	<algo> ²	unique	unique	
sort	<algo>	<algo>	<algo>	sort	sort	(sorted)
count	<algo>	<algo>	<algo>	<algo>	<algo>	count
find	<algo>	<algo>	<algo>	<algo>	<algo>	find
lower_bound	<algo> ²	<algo> ²	<algo> ²	<algo> ²	<algo> ²	lower_bound
upper_bound	<algo> ²	<algo> ²	<algo> ²	<algo> ²	<algo> ²	upper_bound
equal_range	<algo> ²	<algo> ²	<algo> ²	<algo> ²	<algo> ²	equal_range
1varying complexity, 2must be sorted, 3as source					O(1)	O(log n)

multiset	map	multimap	unordered_ set	unordered_ multiset	unordered_ map	unordered_ multimap
↔	↔	↔	→	→	→	→
↔	↔	↔				
	at operator[]				at operator[]	
empty size max_size	empty size max_size	empty size max_size	empty size max_size	empty size max_size	empty size max_size	empty size max_size
			reserve	reserve	reserve	reserve
clear	clear	clear	clear	clear	clear	clear
insert	insert	insert	insert	insert	insert	insert
emplace	emplace	emplace	emplace	emplace	emplace	emplace
emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
erase	erase	erase	erase	erase	erase	erase
swap	swap	swap	swap	swap	swap	swap
<algo> ³	<algo> ³	<algo> ³				
(sorted)	(sorted)	(sorted)				
count	count	count	count	count	count	count
find	find	find	find	find	find	find
lower_bound	lower_bound	lower_bound				
upper_bound	upper_bound	upper_bound				
equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range
O(n)	O(n log n)	O(1) or O(n)	O(1) or O(log n)			

Appendix B

The Author



Torsten T. Will is a C++ expert who has been fascinated by the language since earning his degree in computer science. In addition to C++, he has worked with numerous other languages over the course of his career, including Python, Java, and Haskell. Since 2004, he has contributed his expertise to *c't*, a German magazine for computer technology, where he writes about C++ and Python. In his free time, he enjoys photography.

Index

<code>^</code> -operator (bitwise xor)	101	<code><random></code> (header)	536, 927
<code>_1</code> (placeholders)	977	<code><ranges></code> (header)	654, 834
<code>.</code> -operator (sequence)	107	<code><ratio></code> (header)	935
<code>::</code> -operator (decrement)	75	<code><set></code> (header)	589, 719, 974
<code>::</code> -operator (scope)	66, 235, 314	<code></code> (header)	654
<code>!</code> -operator (negation)	99	<code><sstream></code> (header)	893, 1025
<code>[]</code> -operator (index access)	693, 920	<code><stdexcept></code> (header)	248, 421, 895
<code>*</code> -operator (dereference)	98, 107, 439	<code><stdfloat></code> (header)	132
<code>*</code> -operator (multiplication)	99	<code><syncstream></code> (header)	891
<code>/=</code> -operator (filesystem)	902	<code><system_error></code> (header)	960
<code>/-</code> -operator (calendar)	945	<code><thread></code> (header)	989
<code>/-</code> -operator (division)	122	<code><tuple></code> (header)	628, 912
<code>/-</code> -operator (filesystem)	902	<code><type_traits></code> (header)	495, 964
<code>%</code> -operator (modulo)	122	<code><typeindex></code> (header)	969
<code>++</code> -operator (increment)	75, 104	<code><typeinfo></code> (header)	969
<code>+</code> -operator (addition, positive)	99	<code><unordered_map></code> (header)	766
<code>-</code> -operator (subtraction, negative)	99	<code><unordered_set></code> (header)	751, 914
<code><<</code> -operator	681	<code><utility></code> (header)	910
<code><<</code> -operator (bitshift left)	101	<code><valarray></code> (header)	786
<code><<</code> -operator (stream output)	166	<code>>></code> -operator (bitshift right)	101
<code><=></code> -operator (spaceship)	321	<code>>></code> -operator (stream input)	166
<code><algorithm></code> (header)	175, 264, 588	<code>></code> -operator (greater than)	104, 703
<code><array></code> (header)	167, 347, 495, 583, 698, 853	<code>-></code> -operator (member access)	107
<code><atomic></code> (header)	1043	<code> </code> -operator (logical or)	105
<code><bitset></code> (header)	103, 589, 783	<code> </code> -operator (bitwise or)	101
<code><chrono></code> (header)	358, 552, 751, 930, 996		
<code><cmath></code> (header)	127, 184		
<code><complex></code> (header)	147, 609, 641		
<code><condition_variable></code> (header)	1025		
<code><deque></code> (header)	704, 1025		
<code><execution></code> (header)	802		
<code><filesystem></code> (header)	900		
<code><format></code> (header)	396, 902		
<code><forward_list></code> (header)	711		
<code><fstream></code> (header)	245, 284, 570, 865, 888		
<code><functional></code> (header)	594, 723, 834, 975		
<code><future></code> (header)	1030		
<code><iomanip></code> (header)	127, 300, 416, 704, 868		
<code><iterator></code> (header)	551, 588, 664, 827, 852		
<code><limits></code> (header)	136, 592		
<code><linalg></code> (header)	787		
<code><list></code> (header)	708, 827, 856, 1011		
<code><map></code> (header)	53, 733, 914		
<code><memory></code> (header)	59, 454, 469, 532, 1022		
<code><mutex></code> (header)	1011		
<code><numbers></code> (header)	984		
<code><numeric></code> (header)	175, 541, 699, 834, 1014		
<code><-</code> -operator (less than)	104, 703		

A

Abbreviated function template	186, 565, 582, 813
<code>abs</code>	147
Abstract class	447, 469, 480, 507, 648
Abstract method	58, 202, 477, 507
Access variable	607
Accumulate	699, 834, 1020
<code>accumulate</code>	804
Accuracy	113
Ackermann function	996
<code>acos</code>	149
<code>acosh</code>	149
Actor (OOP)	502
Adapter	670, 779
<code>range</code>	807
Addition	122
Address	116
Address operator	523
advance (iterator)	709
Aggregate	276, 832

- Algorithm 179, 796
 - heap* 829
 - inherently modifying* 820
 - merge* 827
 - nonmodifying* 815
 - Aliasing 124, 189, 523, 525
 - Align 904
 - Alignment 118
 - alloca 838
 - Allocator 665
 - polymorphic* 667
 - Alternative token 106
 - ambiguous_local_time 948
 - Amortized costs 652
 - Anonymous function 602
 - any 982
 - any_cast 982
 - append_range 654
 - arg 147
 - Arithmetic type conversion 123
 - Array 174
 - C 543
 - array (container) 167, 347, 495, 583, 673, 698, 853
 - as_bytes 573, 692
 - as_writable_bytes 692
 - ASCII standard 118
 - asin 149
 - asinh 149
 - Assertion 368
 - Assignment 68, 81, 211, 238, 319, 426, 684
 - compound* 98, 104
 - copy* 1059
 - operator* 427, 430, 452
 - side effect* 76
 - Associative container 319
 - async 1030
 - at 701
 - atan 149
 - atanh 149
 - atomic 1043
 - atomic_flag 1045
 - atomic_ref 1053
 - Atomics 1005, 1040
 - auto_ptr 530
 - auto (abbreviated function template) 186, 649
 - auto (concept) 91, 315, 609
 - auto (structured binding) 210, 702, 913
 - auto (template parameter) 584
 - auto (type deduction) 315, 910
 - Automatic testing 361
 - Automatic variable 526
 - AVL tree 471
 - await_ready 1050
 - await_resume 1050
 - await_suspend 1050
 - Awaitable 1050
- ## B
- back_inserter 800
 - Backend (compiler) 39
 - Backslash 69
 - bad_alloc 253
 - bad_array 253
 - badbit 968
 - Balanced tree 471, 719
 - barrier (class) 1008, 1051
 - Base class 395
 - virtual* 490
 - basic_string 578
 - before_begin 712
 - begin 178, 653, 655, 686
 - Behavior-oriented class 408
 - Bellevue 814
 - bfloat16_t 132
 - Bidirectional iterator 661, 798
 - Big endian 119
 - binary_semaphore 1010
 - Binary operator 415
 - Binary search 826, 851
 - Binary system 100
 - Binary tree 471
 - bind 977
 - bind_back 978
 - bind_front 978
 - binomial_distribution 931
 - Bit 85
 - bitset 103, 589, 783
 - Bitwise arithmetic 98
 - block (statement) 69, 206
 - Bool 86, 141
 - conversion with explicit* 462
 - boolalpha 704, 914
 - boolalpha (stream manipulator) 868
 - Boolean expression 212
 - Boost 971
 - Boost.Filesystem 900
 - Boost.Test 372
 - boyer_moore_horspool_searcher 819
 - boyer_moore_searcher 819
 - Branching 70
 - break (statement) 227

-
- Bubblesort 653
 - bucket_count 756
 - Buckets (hash table) 750
 - Build tool 38
 - Built-in data type 109, 111
 - initialize* 114
 - Byte 116, 126
 - order* 119
 - C**
 - C 55
 - C++ Core Guidelines 1055
 - call_once 1021
 - Callable 597
 - Call by reference 55, 187
 - Call by value 55, 186, 303
 - CamelCase 156
 - Capacity 698
 - capacity (container) 678
 - Capture clause 604
 - C-array 113, 180, 239, 543
 - dynamic* 547
 - Carriage return (CR) 72
 - catch (block) 231
 - Catch-all 57
 - cbegin 686
 - cend 686
 - char (built-in data type) 143, 592
 - char16_t (built-in data type) 144, 642
 - char32_t (built-in data type) 144, 642
 - char8_t (built-in data type) 144
 - Character string 160, 548
 - Character type 113, 143
 - chrono 904, 930, 935, 996
 - chrono_literals 957
 - Civil calendar 944
 - Clang++ 40
 - Class 55, 295, 298, 408, 499
 - abstract* 447, 469, 480, 507, 648
 - enum* 448
 - hierarchy* 404, 1062
 - signature* 507
 - template* 578
 - Class template argument
 - deduction (CTAD) 88, 173, 621
 - clear (atomic_flag) 1045
 - clear (container) 696
 - CLion 43
 - Clock 935, 953
 - clock_cast 956
 - clock_time_conversions 956
 - Closure 842
 - cmatch 918
 - cmp_equal 124
 - cmp_less 124
 - co_await 1046
 - co_return 229, 1046
 - co_yield 229, 1046
 - Code duplication 391
 - Code generation 39
 - Code page 164
 - Collapsing 587
 - Collation 923
 - Collision 750
 - Comma
 - sequence operator* 107, 694
 - variable declaration* 210
 - Command line 41
 - Comment 66, 151
 - Comparator 723
 - compare_three_way 445
 - compareTo (Java) 105
 - Comparison category type 444
 - Compiler 39
 - complex_literals 957
 - Complexity 651
 - Composition 392
 - Compound assignment 98, 104
 - Compound statement 70, 206, 277
 - Compress 832
 - Concept 91, 590, 614, 751
 - Condition 212
 - condition_variable 1025, 1029
 - conj 147
 - const 140
 - const_iterator 356, 551
 - const (keyword) 337
 - Constant 89
 - constant_iterator 649
 - constant_range 649
 - Constant expression 347
 - Constant method 303
 - Constant reference 303
 - const char [] 548
 - const char* 86
 - const-correctness 303, 355
 - consteval 201, 352
 - consteval if 354
 - constexpr 140, 439, 495, 736
 - constexpr (return type) 349
 - constexpr if 215, 615, 856
 - constinit 333
 - Constrained algorithm 659, 795

Constructor 161, 201, 288, 305, 413, 495, 701
 class hierarchies 404
 copy 424, 430, 451, 499, 701
 default 293, 1059
 delegate 292, 1060
 generated 298
 move 432, 452, 701, 1059
 parameters 293
 Container 172, 205, 648, 800, 845
 adapters 779
 associative 319
 classes 669
 flat 716, 779
 ordered associative 320
 pointers 552
 sequence 320
 simple sequence 174
 unordered 972
 contains 669
 Contiguous iterator 662, 798
 continue (statement) 228
 Copy 588, 990
 copy (function) 588
 Copy-and-swap idiom 726
 Copy assignment 1059
 Copy constructor 264, 319, 424, 430, 451, 499, 701
 Copy elision 304, 432, 574
 Copy operator 264
 Core language 92
 Coroutine 229, 1046
 cos 149
 cosh 149
 Count 552
 count 729, 935
 count_if 179
 counting_semaphore 1010
 cout 94
 crbegin 686
 crend 686
 Critical section 1011
 CUDA 788
 Curiously recurring template
 pattern (CRTP) 842
 Curly brackets 154
 current_exception 1038
 Custom data type 300
 Custom deleter 418, 454, 572
 cv-qualifier 357

D

data (method) 688
 data (vector) 888
 Data-holding class 408
 Data race 1005
 Data transfer object 408
 Data type 55
 Data word 116
 Deadlock 1017
 Deallocate 665
 dec (stream manipulator) 870
 Decapsulation 308
 Decay 545, 700
 Decimal system 115
 Declaration 209, 286
 class 277
 forward 192, 286
 function 182
 member default values 291
 variable 68
 decltype 735
 Decrement operator 98
 Deduction guides 622
 default_random_engine 538, 927
 default_searcher 819
 Default constructor 293, 1059
 defaultfloat (stream manipulator) 870
 Default operations 1059
 Definition 80, 207, 286, 412
 function 182
 variable 89
 Degradation (performance) 670
 Delegate (OOP) 1062
 Delegate constructor 1060
 Delegate to sister class 494
 Delegation 292
 Delete
 deleting functions 201, 429, 452
 free memory 109
 Dependency inversion principle 515
 Deprecated 530, 607
 Deque 704, 1025
 Dereference 685
 Dereference operator (*) 98, 439, 523
 Designated initializer 278
 Destructor 202, 264, 412, 451, 1059
 virtual 447
 detach 992, 1037
 Diamond-shaped multiple inheritance 490
 Dice 538
 Digits 592

directory_entry (filesystem) 900
 distance (iterator) 551, 709
 Distribution 929
 divides 975
 Division 122
 do (statement) 218
 Docker 40, 803
 Documentation 152, 568, 1057
 Dot before dash calculation 99
 Double 127, 592
 Double colon 66
 Double word 116
 Doxygen 152
 Drain (software design pattern) 535
 Dummy (unit test) 366
 duration_cast 552, 751, 940
 Dynamic library 39
 Dynamic memory 109
 Dynamic polymorphism 488, 500

E

e (constant) 984
 Eager 659, 1050
 Easter algorithm 302
 ECMAScript 921
 EEXIST 963
 egamma (constant) 984
 element (container) 671
 elifdef (preprocessor directive) 566
 Else branch 213
 Embedded system 961, 1002
 emplace 694
 emplace_back 694
 Empty 668
 Empty statement 79, 208
 Encapsulation 297, 308, 437
 Encoding 164, 642
 end 178, 653, 655, 686
 endl (manipulator) 94, 172
 End of file (EOF) 862
 ends_with 781
 Entropy 929
 enum class 448, 964
 Epoch 957
 Epsilon 132
 equal_range 718
 Equivalence (error condition) 963
 Equivalence groups 923
 Erase 654
 errc 960
 errno 961
 Error 243
 ERROR_ALREADY_EXISTS 963
 error_category 960
 error_code 960
 Error handling 243
 Escape characters 164
 Escaping 69
 Exception 199, 203, 244, 248, 253, 457
 handling 244, 249
 safe 1016
 throw 248
 Exchange 434
 exclusive_scan 804
 Exclusive or 101
 Executable program 39
 execution (namespace) 802
 Execution policy 802
 exp 149
 explicit (keyword) 307
 Explicit type conversion 306
 Exponent 131
 Export 257
 Expression 55, 63, 80, 233
 constant 347
 type 235
 Expression statement 211
 extern 329, 333
 extract 727

F

fabs 127
 Factory (software design pattern) 330, 535
 failbit 968
 Failure 968
 Fake (unit test) 367
 False 86, 141
 fclose 688
 Fibonacci number 996
 file_exists 963
 fill 904
 fill (array) 701
 fill (ios_base) 878
 Filter 949
 final_suspend 1048
 finally (Java) 253
 find 729
 first 692, 910
 First in, first out (FIFO) 704
 fixed (iomanip) 127
 fixed (ios) 870
 Fixture (unit test) 368

Frontend (compiler)	39
Function	55, 67, 281, 581, 594, 723, 975
<i>anonymous</i>	602
<i>body</i>	67, 186
<i>calls</i>	55
<i>declaration</i>	182
<i>definition</i>	182
<i>deleting</i>	201
<i>free</i>	281, 372, 635
<i>global</i>	281
<i>immediate</i>	352
<i>inline</i>	200
<i>member</i>	281
<i>name</i>	185
<i>object</i>	56, 572, 595, 989
<i>overloading</i>	194
<i>parameters</i>	67
<i>pointer</i>	595
<i>string</i>	162
<i>template</i>	186, 564
<i>type</i>	183
function object	599
Function template	578
Functor	600, 840, 990
<i>helper classes</i>	972
functor	597
Future	1033
future_status	1033
fwrite	688

1080 © 2025 by Rheinwerk Publishing Inc., Boston (MA)

Gnu Compiler Collection (GCC) 40
 GobAllocator 665
 goto (statement) 229
 Greatest common divisor (gcd) 834
 Gregorian calendar 944
 Guideline Support Library (GSL) 60, 1056

H

h (hours, literal suffix) 940, 959
 Half word 116
 Handle 457, 569
 has_value 980
 Has-a relationship 391, 475
 Hash 670, 751
 table 749
 Header file 39, 92
 Heap 56, 109, 528, 676
 data structure 829
 Heap-sort 829
 hex (stream manipulator) 870
 Hexadecimal 121, 522
 hexfloat (stream manipulator) 870
 holds_alternative 981
 hours 940
 HUGE_VAL 137
 Hyperthreading 1003

I

i (complex, literal suffix) 959
 IANA time zone database 950
 ICU library 950
 Identifier 64, 72, 235, 237, 634
 compound 237
 if (complex, literal suffix) 959
 if (statement) 69, 78, 212
 if consteval 354
 ifdef (preprocessor directive) 566
 ifstream 169, 245, 570, 888
 ignore 912
 il (complex, literal suffix) 959
 imag 147
 Immediate function 201, 352
 Implemented-by relationship 475
 Implicit type conversion 91, 306, 964
 Import 93, 257
 in_range 124
 Include 39, 66, 92
 guard 287
 path 568
 inclusive_scan 804

Increment operator 98
 Indentation depth 153
 Index access 238, 685, 701
 variant 981
 INFINITY 137
 Infinity 135
 negative 136
 positive 136
 Infix notation 435
 Information hiding 297
 Inheritance 391, 497
 diamond-shaped 490
 multiple 57, 479
 virtual 492
 initial_suspend 1048
 Initialization 64, 68, 209, 426, 457, 701
 constructor 288
 lazy 1021
 list 162, 677
 private data 298
 statement 69
 string type 161
 value 114
 zero 298
 inline 333, 439, 584
 Inline namespace 335, 383
 Inlining 287
 Inner product 804, 833
 Input 94
 input_iterator 699
 input_or_output_iterator 316
 input_range 589, 649, 813
 Input iterator 661, 798, 842
 insert 694
 insert_after 712
 insert_or_assign 734
 insert_range 781
 insert_range_after 712
 Inserter 800
 Instance variable 280
 Instantiate 670
 Instantiate (template) 581
 int_fast16_t 126
 int_least64_t 126
 Int (built-in data type) 85, 592
 int64_t 126
 int8_t 126
 Integer 85
 built-in data type 114
 data type 112
 literal 120
 overflow 123

integral (concept) 610
Intel byte order convention 120
Intel Thread Building Blocks 802
Interface 258, 310, 409, 507
 class 1062
 file 94
 multithreading 1012
Interface segregation principle 511
Intermediate representation (compiler) 39
internal (stream manipulator) 871
Internal partition implementation file 94
International letters 144
inv_pi (constant) 984
inv_sprt3 (constant) 984
inv_sqrtpi (constant) 984
invalid_argument 248, 895
Invocable 751
ios 968
Iota 175, 541, 588, 836, 1014
iota_view 764
is_constant_evaluated 354
is_error_code_enum 960
is_error_condition_enum 960
is_integral 610
is_literal_type 495
is_signed 592
Is-a relationship 391, 497
isfinite 137
isinf 137
isnan 137
isnormal 137
istream_iterator 800
istreamstream 894
Iterator 107, 178, 239, 550, 653, 660
 adapter 800
 bidirectional 661, 798
 category 798
 forward 763, 798, 841

J

Java 55
join 988, 1025
jthread 1051
JUnit 388

K

Kahan method 138
Key (associative container) 671
Keyword 81
knuth_b 930

L

Label 203
Lambda expression 56, 572, 602, 990
 external access 603
last 692, 945
latch 1008, 1051
launch 1035
Launch policy (async) 1031
Lazy 658, 795, 1050
Leap second 956
Least common multiple (LCM) 834
Least significant byte (LSB) 119
left (stream manipulator) 871
Left-to-right (LR) associative 111
lerp (numeric) 834
Letters 144
Lexer (compiler) 39
lexicographical_compare_
 three_way 445, 703
Lexicographical order 703, 913
Library 270, 568
 dynamic 39
Line break 72
Line feed (LF) 72
Linker (compiler) 39, 182, 568
Linux 43
Liskov substitution principle 508
list (container) 673, 708, 827, 856, 1011
Listing 35
Literal 72, 236, 635
 data type 201, 444, 495
 pointer 635
 suffix 957
 user-defined 495, 634, 736, 957
literals (namespace) 781, 910
Little endian 119
LLVM 40
ln10 (constant) 984
Load factor (hash table) 756
local_days 947
local_time 947
locale 903
Locality 847
Lock 457
lock_guard 1011
Lock-free guarantee 1041
log 149
log10 149
log10e (constant) 984
log2e (constant) 984
long (built-in data type) 114, 592

long double (built-in data type) 127, 636
 long int (built-in data type) 116
 long long (built-in data type) 592
 long word 116
 lower_bound 718
 Lvalue 415

M

Macro 39, 555, 561
 type variability 563
 main (function) 76
 Maintainability 498
 make_any 982
 make_error_code 965
 make_error_condition 965
 make_exception 1038
 make_format_args 396, 902
 make_optional 980
 make_pair 736, 910
 make_shared 59, 454, 536
 make_tuple 633, 910, 912
 make_unique 469
 Makefiles 38, 45
 malloc 838
 Manipulator 94, 166, 170, 868, 873, 875
 endl 172
 Mantissa 131
 map (container) 53, 319, 716, 733, 914
 Mapped type (associative container) 671
 match_results 919
 Matrix multiplication 792
 max 132
 max_load_factor 756
 max (duration) 939
 max (numeric_limits) 592
 max (tuple) 910
 mdspan 657, 690
 Member function 281
 Memory 522
 alignment 118
 dynamic 109
 fragmented 849
 uninitialized 838
 memory_order_acquire 1045
 memory_order_release 1045
 Merge 711, 827
 Merge set 827
 Mergesort 827
 Method 55, 161, 184, 280, 281
 abstract 58, 202, 477, 507
 call 184
 Method (Cont.)
 class template 618
 const 339
 constant 303
 inline 286
 pointer 596, 1023
 string 162
 virtual 294, 402, 446, 455, 1062
 Meyers-Singleton 334, 1022
 microseconds, literal suffix 940
 Microsoft Visual Studio 40
 Microsoft Visual Studio Code 46
 midpoint 834
 Milliseconds (<chrono>) 358, 552, 751
 min (chrono) 940
 min (duration) 939
 min (minutes, literal suffix) 959
 min (numeric_limits) 136, 592
 min (tuple) 910
 minstd_rand 930
 minstd_rand0 930
 minus (functional) 975
 minus (subtract) 977
 minutes 940
 Mixed endian 119
 Mock (unit test) 367
 Model-view-controller (MVC) 500
 Model-view-viewmodel (MVVM) 500
 Module 93, 257, 258
 implementation file 94
 modulus 122, 975
 Monadic transformations 56
 Most significant byte (MSB) 119
 move_cast 534
 move_iterator 800
 move (function) 534, 682, 1037
 Move constructor 264, 432, 452, 701, 1059
 Move operator 264, 432, 452
 ms (milliseconds, literal suffix) 940, 959
 mt19937 930
 mt19937_64 930
 Multidimensional span 693
 multimap (container) 716, 745
 Multiple inheritance 57, 479
 Multiplication 122
 multiplies 834, 975
 multiset (container) 716, 740
 Multithreading 987
 Mutable 355, 606, 1012
 mutex 460, 1010, 1011

N

Name	156, 237, 312, 969
Namespace	263, 324
<i>anonymous</i>	327, 342, 570
<i>inline</i>	335, 383
namespace (keyword)	336, 634, 736
nanoseconds (<chrono>)	940
Narrowing	683
Negative infinity	136
Network byte order	120
new (keyword)	109, 528
new (placement)	838
next (iterator)	709
Nibble	116
noboolalpha (stream manipulator)	868
nonexistant_local_time	948
norm	147
not_supported	965
Not a number (NaN)	136
nothrow-new	465
notify_all	1025
notify_one	1025
not-operator (logical negation)	106
now	751, 935
ns (nanoseconds, literal suffix)	940, 959
Null pointer	457
nullptr	525
Number literal	81
numeric_limits	136, 592

O

Object	499
Object-oriented programming	
(OOP)	391, 497
<i>design</i>	500
<i>SOLID</i>	500
Observer (software design pattern)	489
oct (stream manipulator)	870
Octal	121
ofstream	284, 865, 888
once_flag	1021
O-notation	651
Open/closed principle	504
Operand	80
Operator	80, 97, 435
<i>address</i>	523
<i>arithmetic</i>	99
<i>assignment</i>	427, 430, 452
<i>binary</i>	98, 415

Operator (Cont.)

<i>decrement</i>	98
<i>dereference</i>	98, 523
<i>function-like</i>	99
<i>increment</i>	98
<i>logical</i>	98, 105
<i>move</i>	432, 452
<i>pointer</i>	98
<i>postfix</i>	75
<i>prefix</i>	75
<i>relational</i>	98, 104
<i>scope</i>	314
<i>sequence</i>	107, 110
<i>side effect</i>	75
<i>spaceship</i>	321, 601, 703, 754
<i>special</i>	99
<i>ternary</i>	98
<i>unary</i>	97
optimize (regex)	921
Optimizer (compiler)	39
optional	980
ostream	463
ostream_iterator ...	551, 588, 664, 800, 836, 852
ostringstream	893
osyncstream	891, 1029
out_of_range	253
Output	94
output_iterator	316, 661, 798
Overflow	112, 123, 127, 192, 232
overflow_error	253
Overload	235, 345, 564, 579
Override	399
override (keyword)	403
Owner	392, 453, 503, 529

P

packaged_task	1039
pair	672, 910
pairwise_transform	835
par_unseq	802
Parallelism	988
param_type (random)	932
Parameter	55, 71, 185, 585
<i>converting</i>	192
<i>default</i>	196
<i>nontype</i>	625
Parameterized type	87, 173, 578
Parameter-Pack	632
Parentheses	237, 279
<i>missing</i>	560

parse (chrono) 946
 Parser (compiler) 39
 partial_sum 804
 Partition 825
 path (filesystem) 900
 Perfect forwarding 587
 Period (floating point) 134
 phi (constant) 984
 pi (constant) 984
 Pimpl pattern 264
 Placeholder 977
 tie 977
 plus 835, 975
 pmr 667
 Pointer 107, 180, 185, 239, 458, 595
 arithmetic 544
 in containers 552
 literal 635
 method 596, 1023
 null 457
 raw 528, 539, 688
 smart 528, 530, 678
 void 572
 Pointer operator 98
 polar 147
 Polymorphic allocator 667
 Polymorphism 499
 dynamic 488, 500
 runtime 407
 pop_back 696
 Positive infinity 136
 POSIX 966
 Postcondition 218, 1056
 pow 149
 Precedence 110
 Precision 127, 904
 precision (ios_base) 878
 Precondition 1056
 Predicate 815
 prepend_range 712
 Preprocessor 39, 92, 555
 directive 555
 prev (iterator) 709
 print 130
 printf 903
 priority_queue 779
 Program error 244
 proj 147
 promise 1037, 1038
 promise_type 1047
 Proxy object 783
 Pseudo-random number generator 928, 929

ptrdiff_t 126
 public (keyword) 296
 Pure virtual 202, 477, 507
 push_back 694
 put (stream) 885

Q

q-grams 262
 Quadruple word 116
 Quality error 245
 Queue 648, 779
 double ended 674, 704
 Quicksort 653, 825

R

random_device 536
 Random access iterator 661, 798
 Randomness 926
 Range 589, 654, 656, 800
 forward 712
 range_adaptor_closure 842
 range_error 253
 Range adapter 807
 Ranged for 203
 ranges (namespace) 796
 ranlux24 930
 Ratio 950
 Raw pointer 528, 539, 688
 rbegin 686
 read (stream) 887
 real 147
 Real-time system 961
 Receive 1037
 Recursive 996
 recursive_mutex 1023
 Red-black tree 471
 reduce 804
 ref (std) 892
 Refactoring 362
 Reference 59, 185, 239, 343, 524, 738
 collapsing 587
 constant 188
 parameter 291
 Reflection 390
 regex 917
 syntax 921
 regex_constraints 919
 regex_iterator 920
 regex_replace 917
 regex_search 917

regex_traits	919	Scope resolution operator	66
Regular expression	917	second (pair)	910
Rehash (hash table)	756	seconds (<chrono>)	940
reinterpret_cast	488, 1059	Seed (random)	928
Relational operator	104	seekg (stream)	887
Relationship	392, 393	seekp (stream)	890
remove_prefix	164	Semaphore	1010
remove_suffix	164	Send	1037
rend	686	Sentinel	656
replace_with_range	781	seq	802
requires	751	Sequence container	320
Reserve	524, 756	Sequence operator	107
reset	783	set	783
resize_and_overwrite	781	set_exception	1038
Resource acquisition is		set_value	1038
initialization (RAII)	56, 253, 420, 457	set (container)	319, 716, 719, 827
Resource wrapper	253, 418	setf (stream manipulator)	877
Responsibility	503	setfill	300
Rethrow	251, 457	setprecision	127
return_value	1050	Setup (unit test)	369
return_void	1050	setw	300, 416
return (statement)	67, 228, 910	Shadowing	476
Return type	185	shared_ptr	59, 454, 530, 536, 1022
Return value	77	Shared states	366
Return value optimization (RVO)	574, 682	Short	592
Reusability	391, 498	Short circuit evaluation	105, 525
reverse_iterator	800	Short word	116
reverse()	588	Side effects	182, 234, 366
Rewritten candidate	322	Sign	131
right (stream manipulator)	871	Signature	399
Right associative	111	class	58, 477, 507
Rounding	127, 377	signbit	137
Round parentheses	155	signed	114
Rule of five	452	signed_integral	316
Rule of zero	451, 456, 999, 1060	signed char (built-in data type)	143, 592
Runtime	83, 402	Significand	131
polymorphism	407	sin (complex)	149
runtime_error	253, 421	sin (double)	184
Runtime type information (RTTI)	83	Single responsibility principle	502
Rvalue	415, 1066	Singleton (software design pattern)	334, 965
reference	411	sinh	149
		Sink (software design pattern)	535
		size	668
		size_t	124, 699
		sizeof operator	109
		sleep	358
		sleep_for	358, 996
		Slice	791
		Slicing	507
		Smart pointer	250, 528, 530, 678
		smatch	920
		snake_case	156

S

s (seconds, literal suffix)	959
s (string, literal suffix)	959
same_as	751
sample (function)	824
scanf	897
scientific (stream manipulator)	870
Scientific notation (floating point)	134
Scope	68, 314, 324, 412

Social test (unit test)	364	steady_clock	552, 751, 935
Sockets	460	stoi	897
Software design pattern		stol	897
<i>factory</i>	330, 535	stop_token	990
<i>observer</i>	489	strcmp (C)	105
<i>singleton</i>	334, 965	Stream	103, 160, 166, 463, 800, 858, 968
<i>sink</i>	535	<i>buffer</i>	898
SOLID (OOP)	500	<i>manipulators</i>	868
Solitary test (unit test)	364	Stream output iterator	836
sort (function)	314, 588	streampos (ios_base)	890
Space character	71	stride	836
Spaces	153	String	86, 160, 775
Spaceship operator	321, 601, 754	<i>literals</i>	548
Span	649, 690	<i>stream</i>	892
<i>multidimensional</i>	693	string_literals	957
Specialization	783	string_view	164, 291, 657, 690
Splice	674	stringify (macro)	377
splite_after	715	struct	276, 298
Spy (unit test)	367	Structure	290
SQLite	460	<i>type</i>	583
sqrt (complex)	149	Structured binding	702, 913
sqrt (double)	138	Stub (unit test)	367
sqrt2 (constant)	984	STUPID	516
sqrt3 (constant)	984	Subnormal number	135
sregex_iterator	917	subspan	692
ssize	668	subtract	977
Stack	56, 458, 526, 650, 676, 779	Subtraction	122
Standard library	39, 58, 66, 647	Suffix	634
<i>algorithms</i>	800	Suite (unit test)	368
<i>exceptions</i>	253	sv (string_view, literal suffix)	164, 959
starts_with	781	Swap	433, 456
Statement	55, 63, 69, 78	swap (std)	1018
<i>block</i>	78, 206	switch (statement)	222
<i>break</i>	227	sy (string_view, literal suffix)	781
<i>compound</i>	206, 277	Synchronization (thread)	1005
<i>continue</i>	228	sys_days	944
<i>declaration</i>	209	sys_seconds	953
<i>do</i>	218	sys_time	953
<i>empty</i>	79, 208	system_category	960
<i>expression</i>	211	system_error	967
<i>for</i>	219		
<i>goto</i>	229	T	
<i>if</i>	212	Tabulate	53
<i>return</i>	228	Tabulator	153
<i>switch</i>	222	take	836
<i>throw</i>	231	tan	149
<i>while</i>	216	tanh	149
static_cast	109, 488, 965, 1059	Target type (associative container)	671
static (keyword)	329, 336	Teardown (unit test)	369
std.compat (module)	93	tellg (stream)	887
std (module)	93	tellp (stream)	890
std (namespace)	95		

- Template 57, 578, 1065
 alias 627
 class 580, 616
 function 578, 581
 parameter 87
 specialization 638
 typename 580
 variadic 630
Temp value 103, 279, 414, 1066
 reference 411, 587
Ternary operator 98
test_and_set 1045
Test doubles (unit test) 366
Test-driven development (TDD) 363
Testing 285, 361
Text literal 81, 144
Text string 86
Then branch 212
Thread 334, 988, 989
 joining 989
 pool 1002
thread_local 1024
Three-way comparison 105, 321, 601
throw (statement) 109, 231
tie 912
time_t 126
time_zone 947
Time measurement 552, 751, 930, 935
Timeout 1033
Time span 935
Time zone database 949
tmpfile 688
to_chars 130, 897, 984
to_string 130, 820, 896
to_sys 956
to_utc 956
to (ranges) 658, 807
Token 39, 72
 alternative 106
Transform 264, 834, 949
transform_reduce 804
Trivially copyable 1042
True 86, 141
Truth value 113, 141
try_lock_for 1034
try (block) 231
tupe_size (array) 702
Tuple 628, 672, 910
tuple_element (array) 702
tuple_element (tuple) 916
tuple_size (tuple) 916
Two's complement 101
Type 83, 183, 209, 234, 579, 904
 alias 124
 casting 240
 conversion 306, 405, 462, 487, 964
 conversion (arithmetic) 123
 conversion (implicit) 91
 conversions 109
 custom 279, 300
 deduction 315
 exceptions 254
 inference 91, 315
 parameterized 87, 173, 578
 safety 1058
 sink 448
 trait 409
type_traits 495
typedef (keyword) 124, 313
typeid 969
typename 593
Type-trait 916
- ## U
- u16string 164
u32string 164
u8string 164
Ubuntu 43
Unary address operator 523
Unary dereference operator 523
Unary operator 98
Undefined behavior 150, 193, 209, 240, 563
underflow_error 253
Underflow (floating point) 135
Undesirable state 244
Unexpected state 244
unhandled_exception 1048
Unicode 146, 635
Unified initialization 64
uniform_int_distribution 536, 927
uniform_real_distribution 932
Uninitialized memory 838
Union 980
unique_lock 1020
unique_ptr 454, 469, 530, 532
Unit test 363
Universal reference 587, 813, 854
unordered_map (container) 716
unordered_multiset (container) 751
unordered_set (container) 716, 914
Unordered container 972
unseq 802
unsetf (stream manipulator) 877

unsigned_integral 316
 unsigned (built-in data type) 114, 592
 unsigned char (built-in data type) 143
 unsigned int (built-in data type) 116
 unsigned long long (built-in data type) 636
 Unspecified behavior 150
 upper_bound 718
 us (microseconds, literal suffix) 959
 User-defined literal 147, 495, 634, 736, 957
 using (keyword) 336, 476, 615
 using (namespace) 95, 634
 using (type alias) 313
 UTF-8 642
 uz (literal suffix) 699

V

Value 185, 303, 342, 579, 671
 Value initialization 114
 Value-initialized 278
 Value parameter 291
 Variable 67
 access 607
 automatic 526
 condition 1029
 const 341
 declaration 68
 definition 89
 global 191, 205, 284
 instance 280
 local 191
 temporary 279
 Variadic template 630
 vector (container) 176, 319, 673, 679, 845
 vector<char> 781
 Vectorize 802
 vformat 396, 902
 vformat_to 902
 View 656, 807
 views (namespace) 796
 Virtual base class 490
 Virtual inheritance 492
 Virtual machine 56
 Virtual method 294, 402, 446, 455, 1062
 table 1062
 Visual Studio 41, 46
 void* 488
 volatile 357

W

wait_for 1033
 wchar_t (built-in data type) 144
 wcmatch 920
 weak_ptr 530
 wformat_string 903
 while (statement) 216
 Width 904
 width (ios_base) 878
 Windows 46
 Windows Subsystem for Linux (WSL) 43
 Word 116
 write (stream) 887
 wsmatch 920
 wstring 164, 903

X

xUnit (unit test) 368

Y

yield_value 1050
 Yoda condition 420

Z

zero (duration) 939
 Zero initialization 298
 Zero-overhead principle 1067
 zip_transform 821
 zlib 568
 zoned_time 947

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meganf@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: <http://www.rheinwerk-computing.com/5927>. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website <http://www.rheinwerk-computing.com> provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at <http://www.rheinwerk-computing.com>.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2025 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.